Developer-Friendly Test Cases: Detection and Removal of Unnecessary Casts

G. J. B. Vegelien *[1], C. Brandt †[2], and A. Zaidman ‡[3]

[1]TU Delft

Abstract

Test-cube is a tool that focuses on developer-friendly test amplification. *Test amplification* is a technique to improve a test suite by generating new tests based on manually written ones. Currently, these generated tests contain much redundant casting. Our study aimed to improve the readability of these generated test cases by reducing superfluous casting. To test this, we developed multiple cast deleters categorized into two types: simple and fine-grained cast deleters. A simple cast deleter removes casts based on limited knowledge but can make errors. A fine-grained cast deleter only removes casts if it knows they are redundant based on much contextual information. We compared these two types in terms of accuracy by gathering statistical data when running them against real-world examples of amplified test cases. We also discussed the types of casting cases for which they performed well or could be improved based on manual code inspections. In this study, when amplifying all the tests of four public repositories, we found 3,085 casts in 281 tests containing casts. Of these, 97.18% were redundant, and our fine-grained deleter detected and deleted 98.87% of these. We found this fine-grained deleter to be the worthwhile option compared to a simple cast deleter. This was not because the simple cast deleter was slightly less accurate at 97.18% rather than 98.80% but because it showed extremely inconsistent accuracy. The second benefit of the fine-grained deleter was that it caused no tests to fail, while the simple cast deleter caused 18.15% of tests to fail. This paper provides excellent insights and techniques to reduce vast amounts of redundant casting in test amplification. We hope it will make test amplification more developer-friendly and increase its overall practicality.

1 Introduction

Studies show that approximately 25% — up to even 50%[1, 2] — of the software engineering effort is spent on testing[1, 3, 4, 5]. *Test amplification* is a technique that helps reduce this effort. This technique 'mutates the setup phase of existing, manually written test cases and generates new assertions to test previously untested scenarios.'[6]. A tool that implements this technique is DSpot[1].

*g.j.b.vegelien@student.tudelft.nl

†c.e.brandt@tudelft.nl

‡a.e.zaidman@tudelft.nl

[1]https://github.com/STAMP-project/dspot

DSpot works well in terms of added coverage but lacks practicality due to various usability issues[6]. Thus, research was conducted on developer-friendly test amplification[6], and the development of the IntelliJ plugin called test-cube[2] which extends DSpot.

Tests generated by test-cube currently have much superfluous casting. Figure 1 shows an example of this. This study showed that of 3.085 casts in amplified test cases, only 87 were necessary. Other studies show that this problematic for numerous reasons, such as code conciseness, execution speed, and code smells[6, 7, 8, 9, 10, 11].

```
// Original
assertEquals(1066324289, ((int) (((FilterStreamType) (type)).hashCode())));
// Without redundant casts
assertEquals(1066324289, type.hashCode());
```

Figure 1: Example of Casting in Code Generated by Test-Cube

No approaches to reduce the number of redundant casts for amplified test cases have been developed. However, many studies have researched casts and casting analyzers to determine the necessities of casts in different or more general scenarios[12, 13, 11, 8, 14, 7]. These led to various state-of-the-art casting analyzers[12, 7, 14, 15] that can effectively determine the context in which a cast is is not required. The only problem is that only a few of these casting analyzers focus on improving readability. These few are either not built for test code or do not allow false flagging of necessary casts as redundant.

This study aims to achieve better readability for amplified test cases. Our primary hypothesis was that simpler cast deleters might be as beneficial as more advanced deleters if they are allowed to delete some necessary casts. To investigate this concept, we have designed two types of cast deleters and questioned how well each one can detect cast redundancy.

The first type of deleter is the **simple cast deleter**. It deletes casts based on an limited amount of contextual information. This lack of information might cause it to sometimes delete necessary casts. Thus, its ability to detect cast redundancy is measured based on accuracy. Here, accuracy is defined as the sum of correctly deleted casts and correctly retained casts, both divided by the total number of casts. Therefore, the first sub-question is as follows:

SQ1: How much accuracy can a simple cast deleter provide?

The second type of deleter is the **fine-grained cast deleter** which is similar to a traditional casting analyzer. The only difference is that that the sole objective of the deleter is removing redundant casts. This type of deleter does not delete necessary casts and thus requires much contextual information to determine a cast as redundant and delete it. Thus, its ability to detect cast redundancy is not measured based on accuracy, but only regarding the percentage of redundant casting it reduces. Therefore, the second sub-question is as follows:

SQ2: How much redundant casting can be reduced with a fine-grained cast deleter?

Having categorized the cast deleters into two types, we investigate the benefits of both. The main research question is formalized as follows:

RQ: Is a fine-grained cast deleter worthwhile compared to a simple cast deleter in terms of accuracy when simplifying superfluous casts?

[2]https://github.com/TestShiftProject/test-cube

To answer our research and sub-questions we implemented three casting deleters based on preliminary research and ran them against amplified test cases generated from four public repositories. We determined the performance of these casting deleters based on manual code inspections and statistical observations.

This allowed us to make the following contributions:

- three implemented approaches to reduce the number of redundant casts
- insights into multiple types of necessary and redundant casts occurring in amplified test cases
- a study evaluating the advantages and disadvantages of simple and fine-grained cast deleters in terms of accuracy

2 Methodology

To answer the research question we compared multiple statistics generated from a large sample set of 'real-world' examples. Subsequently, we performed a qualitative manual code inspection on a smaller sample set of these 'real-world' examples. This code inspection aimed to better understand the scenarios in which the algorithm is successful or unsuccessful and why.

Section 2.1 and Section 2.2 discus the concept, workings of, and reasoning behind the two types of casting deleters. In the following section we discuss the idea, the workings of, and reasoning behind the two types of casting deleters.

2.1 Simple Cast Deleter

The standpoint that not all amplified test cases must pass provides the opportunity to prototype and test some interesting concepts with varying levels of strictness. Normal cast deleters do not have this ability and leave a cast untouched when the possibility of it being a necessary cast is minimal. We tested the concept of forcing limited information deleters to determine how incorrect they were and how beneficial this error margin could be.

2.1.1 All Cast Deleter

An extreme and interesting concept is deleting all casts. With this approach, we tested the 'loosest' case possible. This provided insights into the maximum loss of necessary casts versus the optimal readable case. Where all its eggs are gathered in the basket of readability and asks, 'what is the worst that can happen?'

This deleter is implemented using the IntelliJ Program Structure Interface (PSI)[3] framework for plugins. The interface enables a simple check whether an expression is a casting

[3]https://plugins.jetbrains.com/docs/intellij/psi.html

expression, and if it is, it deletes it. See Algorithm 1 for the pseudocode of this algorithm.

Algorithm 1: All Cast Deleter
(extends Psi.JavaRecursiveElementWalkingVisitor[4])

```
Input: Amplified Test Cases = $tests$
Output: Test suite Without Any Casting
1 foreach $expression$ in $tests$ do
2     if $expression$ instanceoff $PsiTypeCastExpression$ then
3         $expression$.Remove()
4 return $tests$
```

2.1.2 Double Cast Deleter

```
Assert.assertTrue(((HttpConnection)(HttpConnection)o_cookie__3)).get().hasText());
```

Figure 2: Double casting in amplified test case

Another concept is deleting all but the most outer cast. This algorithm might appear illogical because multiple casts on the same statement are rarely observed. However, as shown in the example in Figure 2, test-cube does generate such cases. According to the Java language grammar[16], double casting is never necessary because casting is essentially a widening or narrowing conversion. That is, the inner cast for double casting is always redundant. Thus, while this algorithm works only for a specific case, it works well for that case, as it will never delete a necessary cast.

To implement this deleter, we also used the PSI framework. As shown in Algorithm 2, we checked all surrounding expressions of a cast to determine whether they were also casts. If so, we deleted them.

Algorithm 2: Double Cast Deleter
(extends Psi.JavaRecursiveElementWalkingVisitor[4])

```
Input: Amplified Test Cases = $tests$
Output: Test suite Without Any Casting
1 foreach $expression$ in $tests$ do
2     if $expression$ instanceoff $PsiTypeCastExpression$ then
3         $children$ = $expression$.getChildren()
4         foreach $child$ in $children$ do
5             if $child$ instanceoff $PsiTypeCastExpression$ then
6                 $child$.Remove()
7                 Repeat from line 3 with $expression = child$
8 return $tests$
```

2.2 Fine-Grained Cast Deleter

Fine-grained casts analysis tools can flag casts as redundant by obtaining more insight into the surrounding contexts of the cast. We examined multiple state-of-the-art casting analyzers. Based on the best one, we implemented a cast deleter.

The 'best analyzer' is defined as the one covering the most casting types/patterns occurring in test-cube. This is *IntelliJ's code cleanup* casting analyzer. Section 2.2.1 provides more information on why this is the case.

[4]https://github.com/JetBrains/intellij-community/blob/master/java/java-psi-api/src/com/intellij/psi/JavaRecursiveElementWalkingVisitor.java

The fine-grained cast deleter works by calling IntelliJ's analyzer for every cast in the test suite. If the analyzer flags it as redundant, the cast is deleted. This is observed in more detail in the pseudocode shown in Algorithm 3.

Algorithm 3: IntelliJ's Cast Deleter
(extends Psi.JavaRecursiveElementWalkingVisitor[4])

```
Input: Amplified Test Cases = $tests$
Output: Test suite Without Any Casting
1 foreach $expression$ in $tests$ do
2     if $expression$ instanceoff $PsiTypeCastExpression$ then
3         $is\_redundant$ = IntelliJRedundantCastUtil.isCastRedundant($expression$)[5]
4         if $is\_redundant$ then
5             $expression$.Remove()
6 return $tests$
```

2.2.1 Rivaling Fine-Grained Cast Deleters

We examined various analysis tools for implementing a fine-grained cast deleter. These tools are not specifically designed for finding redundant casts, and they all work differently. This makes it extremely difficult to compare them with a single measurement. Thus, we grouped the analysis tools into two subcategories: *static analyzers* and non-static *Java verifiers*. This enabled a simple overview comparing the *static analyzers* and discussion of the strengths and weaknesses of the *Java verifiers* that are more difficult to compare.

Static analyzers. We examined four static analyzers: *JDeodorant*[7, 14], *PMD*[17, 18], *JSparrow*[19], and *IntelliJ's code cleanup*[5] We evaluated which was the best based on how many and which casting cases it supported. According to these criteria, *IntelliJ's code cleanup* was the best. *IntelliJ's code cleanup* supported all casting cases that the other analyzers supported and more. This is shown in Table 1.

It is noteworthy that *JDeodorant* does not specifically consider the casting cases represented in Table 1. It can only consider these cases based on all the contextual information it gathers. This is because *JDeodorant* focuses on code smells and refactoring them. One of the code smells is type checking; to refactor the smell, *JDeodorant* gathers type hierarchy information for casts[14, 7].

Casting Cases	***JDeodorant***	***PMD***	***Jsparrow***	***IntelliJ***
Cast to same type as itself	+	+	+	+
Cast when accessing collection elements	-	+	-	+
Widening & Narrowing Cast	+	-	-	+
Cast in Lambda expressions	-	-	-	+
Cast in Conditional expression	-	-	-	+
Boxing & Unboxing Cast	-	-	-	+
Qualifier Consideration of casted types	-	-	-	+

Table 1: Supported Casting of Static Analyzers

[5]https://github.footnote:JavaRecVisitorcom/JetBrains/intellij-community/blob/12245b5e58f629879d1f481e8a0d743de152310e/java/ja
analysis-impl/src/com/intellij/psi/util/RedundantCastUtil.java

Java verifiers. A Java verifier 'ensures that code passed to the Java interpreter is in a fit state to be executed and can run without fear of breaking the Java interpreter.'[16] Code with poor casting can break the interpreter. The general concept behind considering a Java verifier as a foundation for the cast deleter is that the verifier can determine whether deleting the cast would break the interpreter and thereby the test. Note that this means it can only check whether the cast is required for a compiling test and not whether it is required for a passing test.

ESC/Java. **The strength of *ESC/Java*** is that it can determine whether the removal of a cast violates design decisions[12]. Thus, it can determine whether the cast is required for passing or failing a test. **The weakness of *ESC/Java*** is that it requires manually added annotation for the tool to work properly. According to Flanagan et al. [12], this required manual labour is perceived as a heavy burden. Thus, **we have not chosen *ESC/Java*** as a foundation for the fine-grained cast deleter.

JBMC. **The strength of *JBMC*** is that it can verify the absence of uncaught exceptions[15], with a high overall verification rate of approximately 89% for correctly or incorrectly identifying errors[13]. **The weakness of *JBMC*** is that it does not support lambda functions.

We consider *IntelliJ's code cleanup* the best analysis tool. It is the best of all the evaluated static analyzers. Due to its high support for a wide variety of casting cases, it is more suitable than the evaluated *Java verifiers*. The best *Java verifier, JBMC*, works for a much smaller percentage of casting types than *IntelliJ's code cleanup*. *JBMC* does not support casts required for passing a test and casts in lambda expressions.

3 Experimental Setup

This section describes how the experiment is set up and how it can be reproduced. First, we discuss how we obtain the starting data. We then provide a more in-depth overview by writing the complete experiment in pseudocode. Finally, we provide a small section describing the reproducibility of the experiment and how it can be reproduced.

3.1 Data Generation

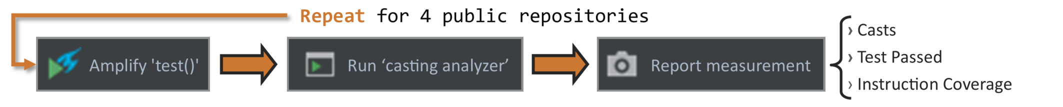


Figure 3: The Data Generation Process

To ensure the data originated from varied real-world examples, we amplified all tests from four different public repositories: jsoup[6], XWiki-commons-filter-api[7], JavaPoet[8], stream-lib[9]. For each public repository, we generated three measurements to benchmark the cast analyzers. The data generation process is divided into three stages, as illustrated in Figure 3.

The first stage amplifies all tests in a public repository using test-cube[2]. All amplified tests that do not contain casts are then filtered out. They are not relevant for our research

[6]https://github.com/jhy/jsoup/commit/ae9a18c9e1382b5d8bad14d09279eda725490c25
[7]https://github.com/xwiki/xwiki-commons/commit/73e1ad74f1ef8a88695b062de6aa25f43604d2b7
[8]https://github.com/square/javapoet/commit/88517888277e3e92cbbdd054228f7f0ff68a841c
[9]https://github.com/addthis/stream-lib/commit/5a3bc87c5314f7771ea3968e9015a3d25536343e

and measuring them would only lead to a false sense of security in terms of the number of passing tests.

The second stage analyzes and removes casts in the amplified test cases using the two types of cast analyzers: two simple cast deleters (Section 2.1) and a fine-grained cast deleter (Section 2.2). This results in a different test suite for every casting analyzer, where the only difference is the casts inside the tests. Everything else in the test suite and environment is be kept constant.

The third and final stage obtains the data. It gathers three different measures of the test suites from stage two: the number of casts, the number of passing tests, and the statement coverage. Additionally, these measures are also obtained for the original test suite (directly obtained from stage 1).

The concept behind these three measurements is that the difference in casts shows the potential improvement, and the test and its coverage work as a reality check on this improvement. The coverage is the extra check that, even if a test passes, it has taken the intended execution path. This can be the case if the cast to a child object is removed and it now executes a similar, but not the same, parent method.

Algorithm 4: Data Generation

```
1  public_repos = {jsoup[6], XWiki-commons-filter-api[7], JavaPoet[8], stream-lib[9]}
2  foreach repo in public_repos do
3      begin — Stage 1: Amplify Tests ——
4          amplified_tests = []
5          foreach test in repo do
6              amplified_tests.addAll( test-cube[2].Amplify(test))
7          amplified_tests = RemoveAllTestsWithoutCasting(amplified_tests) // Tests without cast are
             not relevant for our study.  They would only be a threat to our internal validity, see
             Section 5.2
                                                                                                //
8      begin — Stage 2: Run Casting Analyzer —— (see Section 2)
9          test_suites[0] = amplified_tests
           // (see Algorithm 1:  All Cast Deleter)
10         test_suites[1] = Algo1AllCastDeleter(amplified_tests)
           // (see Algorithm 2:  Double Cast Deleter)
11         test_suites[2] = Algo2DoubleCastDeleter(amplified_tests)
           // (see Algorithm 3:  IntelliJ's Fine-Grained Cast Deleter)
12         test_suites[3] = Algo3IntelliJsCastDeleter(amplified_tests)
                                                                                                //
13     begin — Stage 3: Report Measurements ——
14         foreach test_suite in test_suites do
15             cast_count = IntelliJsPsiStructure[3].CountCast(test_suite)
16             (passing_tests, statement_coverage) = TestRunner[10].RunTests(test_suite)
17             SaveToCSV(cast_count, passing_tests, statement_coverage)
```

3.2 Reproducibility

This research is fully reproducible. The full replication package with instructions on how to run it is available on Zenodo[11]. This paper contains everything required to replicate the same results, from the cast deleters to the amplified and original test methods. For anyone who wants to test it completely, the exact repository snapshots are provided in the footnotes[6,7,8,9].

[10] https://github.com/STAMP-project/test-runner/commit/c84fc308f771b567b24867915066ee116cbeeca5

[11] Zendo doi: '10.5281/zenodo.5035751'

4 Results

The results are divided into two subsections: the statistical data and the manual code inspections. The statistical data section provides a broad view of how well all casting deleters work by presenting the raw and relative measurements. The manual code inspection section describes why they would work well in certain scenarios. It provides insights into these scenarios and discusses how well the casting deleters work with them and how relevant the scenarios are.

4.1 Statistical Data

The data is obtained from four repositories, where the data varies considerably per repository. An example of this is the ratio of casts versus tests. The XWiki repository has more than 35 casts per test on average, while the other repositories have fewer than two casts per test on average. This is shown in Table 2 in the 'No Algorithm' rows. Thus, the data is always represented per repository and as a whole, which is the sum over all repositories.

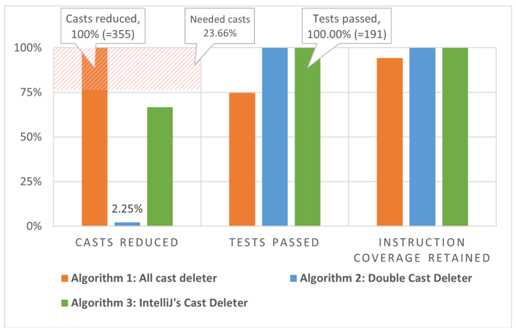
Within some repositories, the requirement for casting is much higher. This is shown in Figure 4, as all casts of repositories (b) XWiki and (c) JavaPoet are redundant. However, the requirement for casting in (a) jsoup and (d) stream-lib is much higher, as more than 20% of all casts are necessary.

IntelliJ's Cast Deleter and the Double Cast Deleter do not delete necessary casts. This is shown in Figure 5; both deleters retained 100% tests passed and 100% statement coverage. However, the All Cast Deleter removes all 2.82% of necessary casts, causing 81.91% of tests to pass and losing 6.33% of statement coverage.

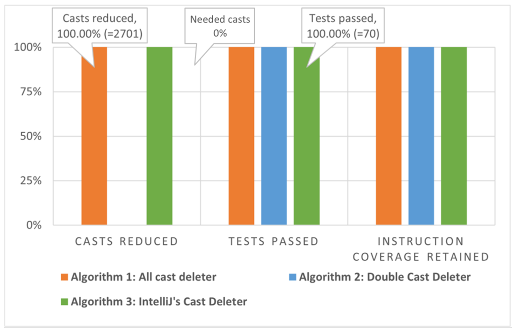
IntelliJ's Cast Deleter has the highest overall accuracy with 98.90%, as shown in Figure 5. The All Cast Deleter was placed second with an accuracy of 97.18%. IntelliJ's Cast Deleter has the higher accuracy because it only reduces 98.87% of the redundant casts opposed rather than 100% reduction of the All Cast Deleter; IntelliJ's Cast Deleter does not delete necessary casts. The Double Cast Deleter was placed third, with an accuracy of 3.08%.

Repository	Casts analyzers	Casts	Tests passed	Instruction Coverage
All	No Algorithm	3085	281	21761
	Alg. 1: All cast deleter	0	230	20316
	Alg. 2: Double Cast Deleter	3077	281	21761
	Alg. 3: IntelliJ's Cast Deleter	121	281	21761
jsoup	No Algorithm	355	191	17992
	Alg. 1: All cast deleter	0	143	16976
	Alg. 2: Double Cast Deleter	347	191	17992
	Alg. 3: IntelliJ's Cast Deleter	118	191	17992
XWiki	No Algorithm	2701	70	233
	Alg. 1: All cast deleter	0	70	233
	Alg. 2: Double Cast Deleter	2701	70	233
	Alg. 3: IntelliJ's Cast Deleter	0	70	233
JavaPoet / stream-lib	No Algorithm	15/14	13/7	2047/1489
	Alg. 1: All cast deleter	0/0	13/4	2047/1060
	Alg. 2: Double Cast Deleter	15/14	13/7	2047/1498
	Alg. 3: IntelliJ's Cast Deleter	0/3	13/8	2047/1489

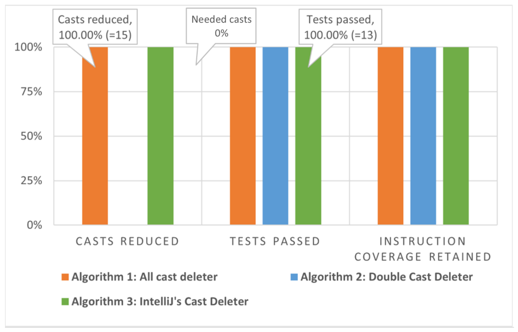
Table 2: Raw Data



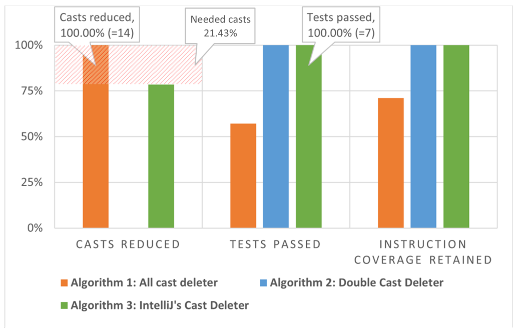
(a) jsoup



(b) XWiki

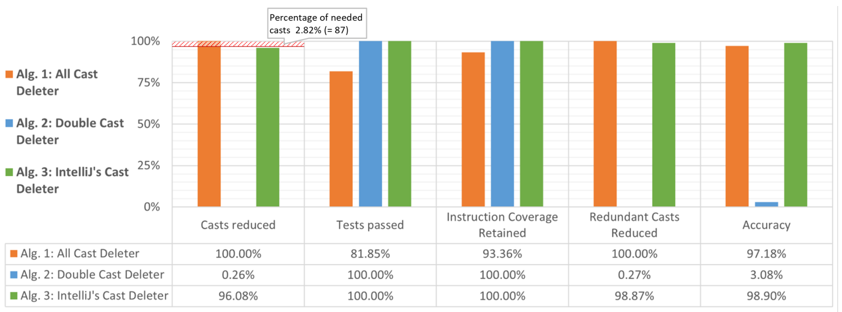


(c) JavaPoet



(d) stream-lib

Figure 4: Benchmarks Results on Each Repository



	Casts reduced	Tests passed	Instruction Coverage Retained	Redundant Casts Reduced	Accuracy
🟧 Alg. 1: All Cast Deleter	100.00%	81.85%	93.36%	100.00%	97.18%
🟦 Alg. 2: Double Cast Deleter	0.26%	100.00%	100.00%	0.27%	3.08%
🟩 Alg. 3: IntelliJ's Cast Deleter	96.08%	100.00%	100.00%	98.87%	98.90%

Figure 5: Total Benchmark of All Algorithms

4.2 Manual Code Inspection

This section describes and categorizes all interesting casting cases. We have defined a cast as interesting if it falls into one of the following categories:

- *Necessary casts*
- *Deleted double casts* by the Double Cast Deleter
- *Redundant casts* not deleted by IntelliJ's Cast Deleter

4.2.1 ***Necessary Casts*** Needed for Compilation

All the necessary casts were also required for the test to compile. No casts caused the test to fail without causing one of the following three types of run time errors:

1. Incompatible types, due to possible lossy conversion between primitive data types.
2. Incompatible types, where conversions between a parent and its subtype can not be done automatically.
3. Cannot find symbol, where there is a call to a method of a child object.

4.2.2 ***Necessary Casts*** in Declaration Statement

The most common necessary casting case is the requirement for a narrowing cast in a declaration statement. Figure 6 shows an example of this cast. This type of casting was required 78 times over 45 tests. It comprised **89.66% of all *necessary casts***. On deleting this type of cast, as does the 'All Cast Deleter', 16% of all tests did not compile.

```
Comment comment = ((comment) (body.childNode(1)));
```

Figure 6: Common Necessary Narrowing Cast

4.2.3 ***Necessary Casts*** in Unnecessary Statement

Many of the casts necessary for the test to compile are located in 'Unnecessary statements'. The term 'Unnecessary statements' refers to statements in an amplified test case that do not affect the test assertions. Figure 7 shows an example of this. This problem of Unnecessary statements in amplified test cases is already a known issue documented in the Developer-Friendly Test Amplification paper by C. Brandt[6]. However, it is interesting that these redundant lines impact casting considerably. After examining all the 87 necessary casts, we found that 76 of these casts occurred in Unnecessary statements. Thus, **87.36% of all *Necessary casts*** occurred in Unnecessary statements.

```
@Test
public void testCommentEndCoverage_mg431_assSep1376() {
    String html = "<html><head></head><body><p>Hello</p></body></html>";
    Document doc = Jsoup.parse(html);
    //The statements in the red lines do not impact the doc.documentType() of doc.
        Therefore these statements are considered to be Unnecessary statements.
-   Comment comment = ((Comment) (doc.body().childNode(1)));
-   Element p = doc.body().child(1);
-   TextNode text = ((TextNode) (p.childNode(0)));
    DocumentType o_testCommentEndCoverage_mg431__13 = doc.documentType();
    Assertions.assertNull(o_testCommentEndCoverage_mg431__13);
}
```

Figure 7: Example of Unnecessary Statements

4.2.4 ***Necessary Casts*** Generated by test-cube

Few necessary casts are newly generated by test-cube. Of the 11 necessary casts in 'Unnecessary statements' (Section 4.2.3), only two casts are not in their original test case. Many casts in the amplified test case also exist in the original test from which they are amplified. These two casts are both narrowing casts required to call a function that is specific for the child.

4.2.5 *Deleted Double Casts*

Every double cast that occurred was similar to the example shown in Figure 2. For all cases where two casts were back-to-back, both these casts were identical. Moreover, the issue occurred with one specific object only: the 'HttpConnection' object from the jsoup.helper package.

4.2.6 *Redundant Casts* Widening to Java's Collection Interface

Many redundant casts missed by IntelliJ's Cast Deleter are the same type of cast. A widening cast from a 'Ellements' object to its supertype, the Java's 'Collection' interface, before invoking the method 'isEmpty()'. Figure 8 shows one such case of the 34 times this type of cast occurred. This casting case comprised **97% of the *redundant casts*** missed by IntelliJ's Cast Deleter.

While a widening cast is generally regarded as a safe cast, it can run through a different execution path if a child class overrides a method. With the cast from 'Elements' to 'Collection', this is the case. Multiple classes exist in the inheritance hierarchy of the Elements class to its supertype the Collection interface. Two of these classes implement the 'isEmpty()' method.

This cast is still considered redundant because both implementations, are deterministically equivalent and part of Java's source code. Deterministic equivalence indicates that they always produce the same results and have no side-effects. Both implementations being part of Java's source code is important because this allows us to assume that the test uses the method to verify something rather than verifying the behaviour of the methods specific implementation.

```
Assertions.assertFalse(((Collection)(docC.getAllElements())).isEmpty());
```

Figure 8: Not Reduced Redundant Collection Cast

4.2.7 Singular *Redundant Cast*

The last singular redundant cast missed by IntelliJ's Cast Deleter is a widening cast from the abstract parent class to its child object initialized by a parser. This test is shown in Figure 9. It is noteworthy that the redundancy of this cast is debatable because the cast might help accurately establish for which child the 'toString()' method is being called. However, we consider it redundant, as it is not required to call the appropriate method. The reason for this is explained in the Java documentation with the following quotation: 'The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behaviour is referred to as virtual method invocation'[16].

```
Evaluator parse = QueryParser.parse(" spangdiv ");
Assertions.assertEquals("spangdiv", ((Evaluator.Tag) (parse)).toString());
```

Figure 9: Not Reduced Redundant Widening Cast

5 Discussion

This section discusses the strengths and weaknesses of every cast deleter, evaluates them against each other, and provides answers to the research questions. It then discusses the reproducibility and validity of this research.

5.1 Discussion of Results

As previously mentioned, all algorithms have strengths and weaknesses. This subsection discusses these and then answers the sub and main questions.

5.1.1 Algorithms

Double Cast Deleter. The general accuracy of the Double Cast Deleter is fairly low. Initially, it might therefore appear the lesser deleter; however, it does not delete necessary casts. This is important, because it means that it works perfectly for the specific case it is designed for. Thus, although this is not particularly beneficial as a stand-alone algorithm, it can work well in combination with other deleters.

All Cast Deleter. However, the All Cast Deleter has a high general accuracy of 97% but has different flaws. The major drawback of this algorithm is that it makes 20% of the tests fail and thereby unusable.

Mitigating the drawbacks of the All Cast Deleter can be achieved in three ways, with the restriction of keeping the Deleter 'simple':

- Cross-check with the original method to leave in all casts not generated by test-cube; see Section 4.2.4.
- Revert all statements to where a compile-time error is raised; Section 4.2.1.
- Add an exception to the deletion of casts when the cast is required for narrowing casting in a declaration statement. This will solve 88.24% of failing tests; Section 4.2.2.

IntelliJ's Cast Deleter. This cast deleter performed extremely well and removed almost every redundant cast. It had the highest accuracy of all three deleters and even held this position for each individual repository. This fine-grained form allows it to handle a wide variety of casting cases, while the simple cast deleters mainly perform well for a single and simple case.

This deleter can be further improved by adding a simple check for the specialized case of a widening cast to the Java 'Collection' interface. This would increase the accuracy to approximately 100% for our tested set. It is noteworthy that this type of redundant casting can still occur within different inheritance hierarchies. We call this generalized type of cast a 'Generalized Redundant Collection Cast' (GRCC).

Deleting all GRCCs is significantly more difficult than only deleting the previously mentioned specific redundant Collection-type casts. With the specific Collection-type cast, we know that both implementations of the 'isEmpty()' method are equivalent Section 4.2.6. However, we do not know this for any other method. Thus, deleting all GRCCs requires determining whether two functions are functionally equivalent. This is generally considered an unsolvable problem[20, 21]. Therefore, improving this deleter to reach an accuracy of 100% might be impossible. However, this is a universal problem that holds for other deleters.

This deleter has two non-universal drawbacks. It is a fine-grained deleter, so it probably takes longer to run than the simple deleters. Secondly, it is based on the IntelliJ PSI structure; thus, it only works in the IntelliJ Integrated Development Environment (IDE). This is not a major drawback, as one purpose of this research was to improve the IntelliJ plugin test-cube.

5.1.2 Research Question

SQ1: *How much accuracy can a simple cast deleter provide?* The accuracy of the most accurate simple cast deleter, the All Cast Deleter, is 97%. However, the accuracy of this deleter is relatively context-dependent and can vary considerably. The accuracy is, for example, only 76% for the tests of the jsoup repository, as the need for casting is much higher.

SQ2: *How much redundant casting can be reduced with a fine-grained cast deleter?* The fine-grained cast deleter has eliminated 98.87% of the total number of redundant casts. It could potentially reach 99.9% by adding a simple check for the edge case of widening casts to the Collection object. However, this is hypothetical, as we have not studied the frequency of similar casts, such as the exceptional cast described in Section 4.2.7 or the possibility of GRCCs described in Section 5.1.1.

RQ: *Is a fine-grained cast deleter worthwhile compared to a simple cast deleter in terms of accuracy when simplifying superfluous casts?* The fine-grained cast analyzer is the better option. Although the difference in the general accuracy between the more accurate simple cast deleter and IntelliJ's fine-grained Cast Deleter is below 2%, the benefit of causing no test to fail is huge.

The fine-grained cast deleter might not be worthwhile for two reasons. Firstly, a significant difference might exist between the runtimes of the deleters. Empirically speaking, we have noticed no difference. However, no real evidence supports this claim. Secondly, we might be unable to rely on IntelliJ. In this case, a huge amount of added investment costs would be required, outweighing the benefits over the simple cast deleter.

5.2 Threats To Validity

Some factors might threaten the validity of this research. We have taken measures to mitigate these. The following sections explain these threats, explain how they are mitigated, and debate whether a risk remains.

Consistency. A threat to the validity of the obtained data is its consistency. One measurement used for the data is the number of passed tests. While test outcomes should always be fully deterministic, a flaky test can occur.

In our project, we ran many different categories of tests, with network tests being one of them. This can be an issue as, according to a 2014 study by Qingzhou Luo[22], network tests are a common category for flaky tests. We want to cover all types of tests. Therefore, we cannot simply disregard tests with a higher probability of being flaky. However, this might impact the consistency of the data. To mitigate this risk, we repeated all tests 10 times and observed no change in the number of passed tests. This does not indicate that there are no flaky tests. Previous studies have concluded that the probability of an unreliable test failing 10 consecutive times is approximately 35% (=0.9^{10})[23], and that of it passing five consecutive times is 77%[24].

Internal Validity. We found some threats to the causal relationship between the treatment and outcome. The measurements obtained might not accurately represent the effects of the cast deleters. Therefore, two measurements were taken.

A statement coverage was added as a measurement to ensure that a passed test tested the methods as intended. Due to the removal of a cast, it could call its parent method. In this scenario, a necessary cast is removed. The only problem here is that the test might still pass, and we would not know that a necessary cast was removed. This added measurement reduces the risk of wrongly stating that a cast was correctly reduced. However, it does not fully solve the issue. When the child method has the same number of statements, the removal of a necessary cast goes under the radar.

To ensure the measurement of ‘tests passed’ has a representative meaning when below 100%, we reduced the sample set to only consider tests that use casts. If this was not the case and we merely ran all amplified tests, the test passed would not be representative. Consider a scenario where a cast deleter caused every test method with a cast to fail. The ‘tests passed’ would then only show how many test contain a cast and not how well the cast deleter worked.

External Validity. Projects can vary considerably in terms of their casting requirements[8]. Thus, we have based our results on four different public repositories. While this remedy makes the study more generalizable to all different real-world cases, the results might still differ in some scenarios.

Conclusion Validity. A threat exists to the ability to draw correct conclusions concerning relationships between treatments and outcomes. We cannot be certain how many casts were missed or wrongly deleted by only examining the statistical data. Estimation is possible by considering the number of passed tests; however, the true backing of the data originates from the manual code inspections.

6 Conclusion

We found 97% of casts were redundant of the 3,085 examined casts in the 281 amplified test cases. The fine-grained cast deleter, based on the IntelliJ casting analyzer, is the best option for two reasons. First, it has an accuracy of 98.89% compared to the highest accuracy of 97% a simple cast deleter has reached. Although the difference is less than 2%, the consistency of the accuracies is essential. When disregarding repositories containing only redundant casts, the difference in accuracy would be approximately 15%. Second, the fine-grained cast deleter has not caused any test to fail, while the simple cast deleter caused 18.15% of tests to fail. However, in the scenario where the deleter cannot be IntelliJ-specific, the development costs may outweigh the mentioned benefits.

Future research can be conducted to improve the accuracies of both types of casting deleters. Furthermore, the readability can be improved by removing redundant brackets. The amplified test case is generated with brackets surrounding the cast and the object or statement it is casting. The current tools remove only the brackets of the cast and not those of the object or statement.

By researching and developing cast deleters with astounding accuracies up to almost 99%, we can say to have tackled one of the issue of amplification testing. This makes amplification testing a more pleasant and streamlined process.

We have tackled superfluous casting with an accuracy of 99%, making amplification testing more developer friendly.

References

[1] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering*, vol. 45, no. 3, pp. 261–284, 2019.

[2] I. Sommerville, *Software engineering.* Pearson/Addison-Wesley, 2004.

[3] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), p. 179â190, Association for Computing Machinery, 2015.

[4] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing," *Empirical Software Engineering*, vol. 11, no. 1, pp. 5–31, 2006.

[5] R. S. Pressman, *Software engineering: a practitioners approach.* McGraw Hill., 5 ed., 2000.

[6] C. Brandt and A. Zaidman, "Developer-friendly test amplification," 2021.

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

[8] L. Mastrangelo, M. Hauswirth, and N. Nystrom, "Casting about in the dark: An empirical study of cast operations in java programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–31, 2019.

[9] J. Winther, "Guarded type promotion: eliminating redundant casts in java," in *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, pp. 1–8, 2011.

[10] P. OâHear, "Analysing the use of casting in java systems," 2019.

[11] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 507–518, IEEE, 2017.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Pldi 2002: Extended static checking for java," *SIGPLAN Not.*, vol. 48, p. 22â33, July 2013.

[13] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "Jbmc: A bounded model checking tool for verifying java bytecode," in *Computer Aided Verification* (H. Chockler and G. Weissenbacher, eds.), (Cham), pp. 183–190, Springer International Publishing, 2018.

[14] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *Journal of Systems and Software*, vol. 83, no. 3, pp. 391–404, 2010.

[15] L. Cordeiro, D. Kroening, and P. Schrammel, "Jbmc: Bounded model checking for java bytecode," in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Beyer, M. Huisman, F. Kordon, and B. Steffen, eds.), (Cham), pp. 219–223, Springer International Publishing, 2019.

[16] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st ed., 2014.

[17] D. Dixon-Peugh, M. Griffa, P. Herlin, T. Copeland, J. Patel, A. Ezust, O.-M. Mork, M. Griffa, R. Kubacki, T. Slota, and et al., "Pmd code style rules." `https://pmd.github.io/latest/pmd_rules_java_codestyle.html#unnecessarycast`, May 2021.

[18] D. Dixon-Peugh, M. Griffa, P. Herlin, T. Copeland, J. Patel, A. Ezust, O.-M. Mork, M. Griffa, R. Kubacki, T. Slota, and et al., "Pmd error prone rules." `https://pmd.github.io/latest/pmd_rules_java_errorprone.html#classcastexceptionwithtoarray`, May 2021.

[19] "Remove redundant type casts." `https://jsparrow.github.io/rules/remove-redundant-type-cast.html#limitations`, Nov 2014.

[20] V. A. Zakharov, "The equivalence problem for computational models: Decidable and undecidable cases," in *Machines, Computations, and Universality* (M. Margenstern and Y. Rogozhin, eds.), (Berlin, Heidelberg), pp. 133–152, Springer Berlin Heidelberg, 2001.

[21] M. Kessel and C. Atkinson, "On the efficacy of dynamic behavior comparison for judging functional equivalence," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 193–203, 2019.

[22] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 643â653, Association for Computing Machinery, 2014.

[23] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?," in *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 492–502, 2020.

[24] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 433–444, 2018.