An Empirical Evaluation for Object Initialization of Member Variables in Unit Testing

Stefan Fischer
Software Competence Center Hagenberg GmbH
Hagenberg, Austria
stefan.fischer@scch.at

Evelyn Nicole Haslinger
Symflower GmbH
Linz, Austria
evelyn.haslinger@symflower.com

Markus Zimmermann
Symflower GmbH
Linz, Austria
markus.zimmermann@symflower.com

Hannes Thaller
Institute for Software Systems Engineering
Johannes Kepler University
Linz, Austria
hannes.thaller@jku.at

***Abstract*—Automated test case generation techniques usually aim to maximize some coverage criteria. For object oriented languages, like Java, the branches that can be reached in source code, frequently depend on the internal object state. Meaning certain branches will only be taken, if fields inside the tested class are set to specific values. It is however not obvious how much of the internal object state can be controlled. In this paper, we analyzed a corpus of 110 open source systems to evaluate how settable their classes are. Meaning we looked for ways that fields inside classes can be written. For instance, we analyzed the source code to identify setter methods that can be used to set the values of a field. Our results show that 66.5% of fields can be set to a desired value, while 31.5% of fields may be only settable to particular values or require a more in depth analysis. Only for 2% of fields, we did not find any way to set their values.**

***Index Terms*—test automation, test case generation, object state, test setup**

I. Introduction

Automated test case generation has been one of the most active research topics in the past decades. From random testing over model-based testing and combinatorial testing over to structural testing using symbolic execution and search based testing, a wide range of different techniques have been developed [1]. All these techniques usually aim to maximize the coverage (e.g. statement or branch coverage). In case of procedural languages, it may be enough to find the correct values that function parameters have to be set to. When testing object-oriented languages, like Java, we may need to make sure the class under test (CUT) has the correct internal state for each test to achieve high coverage [2], [3]. One way to set the internal state of an object is to use Java reflection and simply set all fields to the required values. However, this is not the way the tested classes are intended to be used and not how a developer would normally use them. A more realistic way of setting the object state is using the interface provided by the class, like setters or constructors. Moreover, when using reflection one might violate implicit class invariants, which might make sure that values for fields make sense semantically [4]. For instance, fields might only have a specific range of values for them that are valid, or multiple fields might have implicit consistency rules between them, so values among these fields are only valid if they adhere to these consistency rules. Shamshiri et al. found that using reflection can lead to false positive tests for fault detection, in their experiment about comparing different test generation approaches [5].

In this paper, we analyzed the source code of a corpus of 110 open source projects to see how many of the fields in their classes can be directly set to a desired value [6]. Using the interface of a class to set the fields to the values we want for testing, makes generated tests more like a human would use the class and develop tests for it. We found that 66.5% of fields could be set to a freely selectable value that can be either assigned directly (e.g. public fields) or passed as a parameter through a setter or constructor of the class. Moreover, for some of the setters and constructors our simple analysis could not decide if a field could be set to a desired value, because the method bodies were too complex (i.e. contained loops, throw an exception, ...). Therefore, there might be even more fields that could be set, but to recognize that correctly a more sophisticated analysis is required (e.g. symbolic execution).

II. Internal Object State

We define the internal state of an object as the values that the fields in the class are set to. For testing, we might want to change the internal state to reach different branches within methods and achieve a higher code coverage. However, it might not always be possible to set certain fields to the desired values. As an example, consider the source code snippet for a stack of playing cards in Figure 1. If we first look at the implementation of class `Card`, we can see that we can set the two fields `suite` and `value` to any arbitrary integer values through the constructor. Furthermore, field `value` can be even set later on with a setter call. Therefore, we can reach all branches in method `compare` during testing, by simply setting the field values of the two `Card` objects accordingly.

A more complex example is class `CardStack`, for which the constructor initializes the stack with the maximum number of cards it can hold through parameter maxSize. The remaining two fields are initialized, but not directly set to values we can choose. These fields can only be manipulated by calling push or pop on the stack. Hence, field size can only assume values between 0 and maxSize, because other values for it would be invalid and the push and pop methods do not allow them. If we used reflection, we could set size to any integer value and therefore force a test to enter a particular branch. However, this would not be a realistic use of the CardStack class and would potentially violate the semantics of the class. Moreover, setting field size to an arbitrary value could potentially break the code and produce a failing test, in a way that would never happen in real usage. For instance, if we set size to a negative value and then try to perform a push, we would produce an exception that is based on an object state that can not be reached when using the class without reflection. Hence, the produced test case is not realistic.

```java
public class CardStack {
  private int maxSize;
  private Card[] stackArray;
  private int size;
  public CardStack(int maxSize) {
    this.maxSize = maxSize;
    this.stackArray = new Card[maxSize];
    this.size = 0;
  }
  public boolean push(Card c) {
    if(size>=maxSize)
      return false;
    stackArray[size++] = c;
    return true;
  }
  public Card pop() {
    if(size<=0)
      throw new EmptyStackException();
    return stackArray[size--];
  }
}
public class Card {
  private final int suite;
  private int value;
  public Card(int suite, int value){
    this.suite = suite;
    this.value = value;
  }
  public void setValue(int value){
    this.value = value;
  }
  public int compare(Card other){
    if(this.value != other.value)
      return this.value - other.value;
    return this.suite - other.suite;
  }
}
```

Fig. 1. Code example of stack of cards

III. Experiment

We developed a lightweight static code analysis for Java that tries to identify which fields can be set to specific values. The simplest way the internal state of an object can be manipulated is with *public* fields, meaning they can be directly accessed and overwritten with a desired value. Similarly, we can directly access *protected* and *package-private* fields, if we generate the test setup in the same package as the CUT.

For fields that are *private*, and therefore can not be simply set to a desired value, it may be possible to set them via a constructor or a setter method. Our analysis identifies all non-*private* constructors and methods (i.e. potential setters) that write a value to a field. Furthermore, our analysis can differentiate between different values that are written to the field, inside the setter or constructor. We refer to the different value types as:

- **Parameter**: If a parameter of a setter or constructor is directly assigned to a field, without being altered inside the method. Hence, we can set the field to any desired value. Therefore, the field is fully settable.
- **Constant**: If a setter or constructor writes a constant value to a field. Hence, we can set a field only to a specific value, which is useful if a certain test requires the field to be at exactly this value.
- **Conditional**: If a setter or constructor has a simple condition, for instance to make sure the value of an argument is in a specific range. Our analysis is able to interpret very simple comparison conditions only. Simple relational operators are implemented to either compare the values of two variables or one variable to a constant.
- **Multiple**: If a setter or constructor writes a field multiple times. For instance, if several branches of an if-statement write the field, but the condition is too complex for our analysis to interpret.
- **ComplexType**: If a setter or constructor writes a new complex type to a field. For instance, inside a constructor a statement sets a field to a new object by calling a constructor of a class (e.g. o=new Object();).
- **Other**: Covers any other value type we did not specifically distinguish. For instance, if a field is written with a more complex expression than a simple assignment. A statement like field=parameter+1; would already fall in this type.
- **TooComplex**: If a setter or constructor has a structure that is too complex for our analysis to interpret. For instance, if a method contains a loop, a switch statement, an anonymous class, calls itself recursively, calls another method that is too complex, or has to deal with exception handling our analysis marks the method as too complex to handle for the current analysis.

When applying our analysis to the CardStack example in Figure 1, we find that field maxSize can be set to *Parameter* by the constructor. The same constructor sets field size to a *Constant* value and stackArray to *Other* in our classification. Moreover, the analysis recognized that method push writes field stackArray, but because it is only written in one branch of the method, it is recognized as *Multiple*. Class Card on the other hand is easier to set up correctly, because the constructor sets both fields to a *Parameter* value. Furthermore, the setter method setValue sets field value

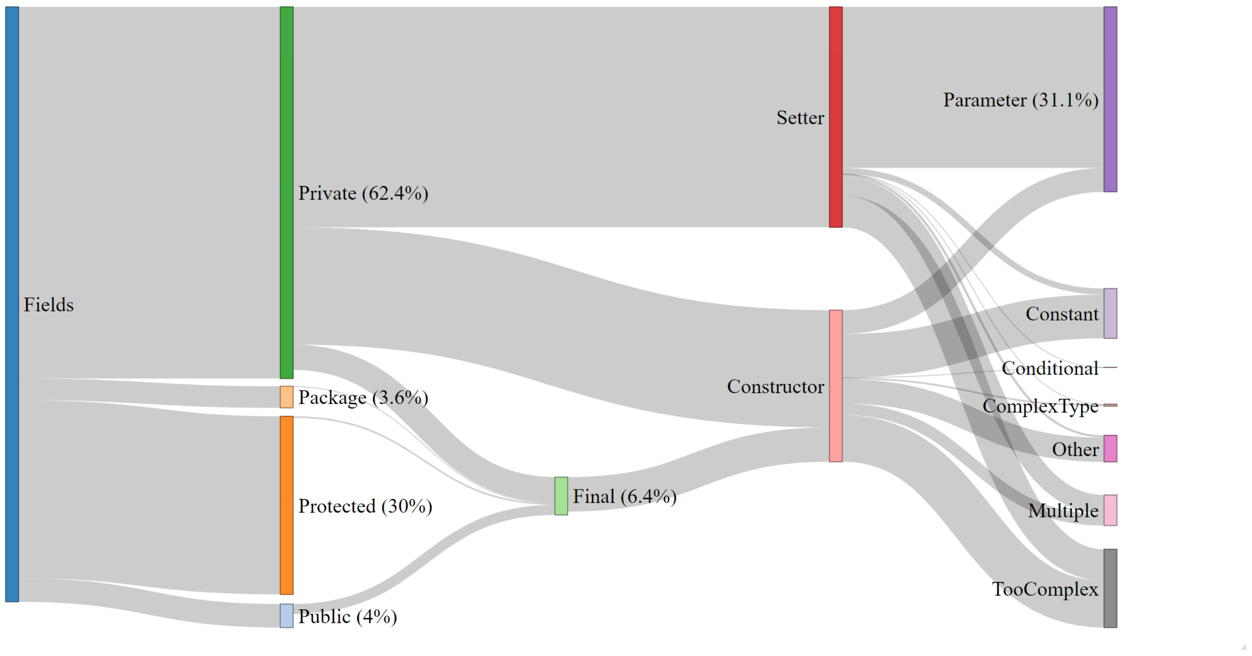

Fig. 2. Overview of the settability of all analyzed Fields

also to a *Parameter* value. Setter methods have the added advantage over a constructor that they can be called on an existing object. Hence, if the value of a field is overwritten, for instance when trying to set another field, the setter can be called to write the field back to the desired value. Overall, the internal state for an object of class `Card` is 100% settable.

With this analysis, we measure how many fields inside classes could easily be set to different values and how freely we can influence the internal object state. We used the SF110[1] benchmark by Fraser et al [6]. This benchmark is a collection of 110 open source projects from SourceForge containing 23,886 classes and more than 6.6 million lines of code.

IV. Results

We analyzed the SF110 corpus and found 18,857 classes that contained 146,467 fields, for which we checked if and how their values can be influenced. In our analysis, we excluded classes that did not contain any fields.

Figure 2 gives an overview of the 146,467 fields and how they could be set. All fields that are not *private* and not *final* (in our results 3,317 *public*, 43,474 *protected*, and 5,113 *package-private* fields) can be set by any class within the same package. Therefore, if the test setup is generated in the same package as the CUT these 51,904 fields (35.4% of all fields) can be directly assigned to the desired values to maximize coverage. The remaining 85,256 non-*final* but *private* fields can potentially be set via a setter method or a constructor. For the presentation in Figure 2, we first looked for fields that could be set to a *Parameter* by a setter or if no setter was found by a constructor. Since, this type of plot is not meant to display overlaps, we subsequently only display the fields that could not be set to a value already. Therefore, we checked for each field if a setter or constructor sets them to a value type, in the following order: *Parameter*, *Constant*, *Conditional*, *Complex type*, *Other*, *Multiple*, *TooComplex*. For influencing the internal object status, we prefer setters over constructors, since they can be called later on in a code sequence to correct fields that may have been overwritten to wrong values. Hence, for each value type we first look for a setter and then for a constructor that could set the field. We found a setter that assigns the field directly to a parameter for 39,666 (46.5%) of the non-*final* but *private* fields. For a further 4,086 of the private fields we found a constructor that sets them to a parameter. *Final* fields can only be set directly hard coded at declaration or by a constructor. We found a constructor that sets *final* fields to a *Parameter* for 1,767 of the 9,307 *final* fields. In sum we could set 97,423 (66.5%) fields to a desired value, via direct assignment (i.e. *public*, *protected*, or *package-private* fields) or a setter or constructor that sets them to a *Parameter*. For 46,085 fields (31.5%) we found a setter or constructor that would set them to one of the other value types, meaning there might be a way to set them, but our analysis could not determine it. Some fields might be settable only to certain values (e.g. constants, or limited ranges by conditions), or a more sophisticated analysis is required to determine if a field could be set to the desired value at the time. Finally, for 2,959 (2%) of fields we did not find any way to set them. These fields are either initialized at declaration and never overwritten, or only written by *private* setters or constructors.

[1]http://www.evosuite.org/experimental-data/sf110/

Next we investigated how many fields are settable (i.e. directly assignable or can be set to a *Parameter* by a setter or constructor), per class and per system. In Figure 3 we depict the rate at which fields inside classes and systems can be set. On average 50.3% of fields in classes could be set. For 7,659 (40.6%) classes all fields could be set. On the other hand, for 4,096 (21.7%) classes none of the fields could be set, with only our simple analysis. When looking for fields over entire systems we found that on average 57.4% of fields in a system could be set. Only for four of the systems, all fields could be set. Similarly, we depicted the rate in which classes within the 110 systems could be fully set (i.e. all their fields could be set) in Figure 4. We found that for an average of 44.7% of classes within a system, it is possible to set all their fields.

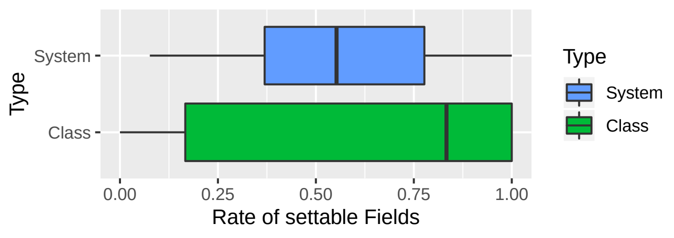

Fig. 3. Rate at which Fields are Settable over Systems and Classes

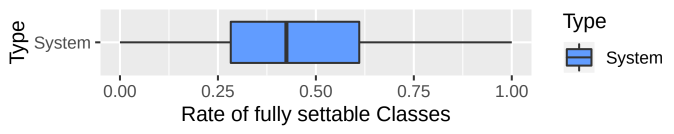

Fig. 4. Rate at which Classes are Fully Settable over Systems

We analyzed 23,312 constructors, of which 3,889 (16.7%) were too complex for our analysis, and 96,276 potential setter methods, of which 13,897 (14.4%) were too complex. Therefore, a more sophisticated analysis will potentially be able to detect ways to set many of the 46,085 fields where we could not decide if they are settable or not.

V. Threats to Validity

The SF110 corpus that we analysed for this paper is a collection of 110 Java projects randomly selected from SourceForge. Thus, it provides high confidence in the possibility to generalize our results to other open source software.

Threats to internal validity might come from how the experiment was performed. The analysis was tested with some smaller examples and we did random checks to confirm the result data on some classes. Moreover, our analysis does not consider possible side effects that could appear where the setting of one field might influence the values of other fields. Such cases might lead to problems when trying to set multiple fields at once for certain tests, which are not expressed in our data.

VI. Conclusions and Future Work

We analyzed 110 Java projects and searched for possible ways to set 146,467 fields in 18,857 classes to control the internal status of an object. Our results showed that, even with simple static code analysis, we could find ways to set 46.5% of the private fields. The 51,904 non-private and non-final fields could be set directly by generating the test setup in the same package as the CUT. With our analysis, we found that 66.5% of the fields could be set to a desired value. A further 31.5% of the fields might be settable with a more sophisticated code analysis. Finally, only for 2% of all the fields we did not find any possible way to set them.

For our future work, we plan to further improve and apply our analysis to generate code sequences that can create and set Java objects with the desired internal state. This will allow us to execute the code sequences and evaluate if all fields within the objects are actually set to the correct values. These sequences could be directly used for test case generation. Another item for our future work is to identify which fields are even relevant for test case generation. Not all fields might be relevant to maximize code coverage. For instance, only some fields in a class might be part of conditions to enter or not enter a branch. Knowing which fields are relevant for different test cases is important for high coverage test generation and will allow us to assess for how many tests setup sequences can be generated. The main goal for our future work is to generate valid test cases without reflection, which resemble how a human would write such test cases.

Acknowledgments

The research reported in this paper has been supported by the Austrian ministries BMVIT and BMDW, and the Province of Upper Austria in terms of the COMET - Competence Centers for Excellent Technologies Programme managed by FFG.

References

[1] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, “An orchestrated survey of methodologies for automated software test case generation,” *Journal of Systems and Software*, pp. 1978–2001, 2013.

[2] P. McMinn and M. Holcombe, “The state problem for evolutionary testing,” in *Genetic and Evolutionary Computation - GECCO 2003, Genetic and Evolutionary Computation Conference, Chicago, IL, USA, July 12-16, 2003. Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 2724. Springer, 2003.

[3] A. Arcuri, G. Fraser, and J. P. Galeotti, “Automated unit test generation for classes with environment dependencies,” in *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*, 2014.

[4] K. R. M. Leino and P. Müller, “Object invariants in dynamic contexts,” in *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, 2004.

[5] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, “Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (T),” in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015.

[6] G. Fraser and A. Arcuri, “A large-scale evaluation of automated unit test generation using evosuite,” *ACM Trans. Softw. Eng. Methodol.*, 2014.