NullTerminator: Pseudo-Automatic Refactoring to Null Object Design Pattern

Ştefan Medeleanu and Petru Florin Mihancea
LOOSE Research Group
Politehnica University of Timişoara, Romania
Email: stefan.medeleanu@gmail.com, petru.mihancea@cs.upt.ro

***Abstract*—Restructuring legacy code to improve its structure and understandability is difficult and adequate tool support is required. While the advantages of the *Null Object* pattern are widely recognized, the first tool support has only recently emerged. We complement it with NULLTERMINATOR, a prototype tool to assist developers in the instantiation of the *Null Object* design pattern in Java programs. We describe the main functionalities of the tool and some important internal details on an accompanying example. The demo concludes presenting some initial results.**

***Keywords*-refactorings; design patterns; Null Object**

I. INTRODUCTION

A large part of software development lifecycle is represented by maintenance. The effort spent in this phase depends on many different factors one of them being the ease of understanding the source code of the maintained program.

Frequent and/or large conditional statements are often considered as an obstacle for source code comprehension e.g., [1]. In particular, the 'null' checks (e.g., *if(var!=null)*) are seen as raising accidental difficulties for object-oriented code understanding because they can be avoided by employing the *Null Object* design pattern [2],[1].

However, transforming legacy code in order to remove such problems is not trivial and thus, tool support is needed to assist developers. Many modern IDEs such as ECLIPSE [3] have already implemented restructuring engines for automatic execution of many well known refactorings described in [1]. JDEODORAT has complemented them by providing support to replace with polymorphic calls the conditional checks verifying the concrete type of an object [4]. Lately, Gaitani et al. [5] implemented the first approach for automatic restructuring of Java code to *Null Object* pattern. The underlying methodology is dedicated for the special case of optional fields (i.e., object properties that are not always initialized with a non-null value) probably because of the difficultly to preserve the code behaviour. However, as suggested in [6], the *Null Object* pattern can be used in the context of eliminating problems related to methods returning the 'null' value. Consequently, applying the pattern in this context must also be considered.

In this demo we introduce NULLTERMINATOR[1], an ECLIPSE plugin prototype assisting developers introducing the *Null Object* pattern in Java programs. While it is more general because it is not dedicated only for the case of optional fields, the cost is that the tool is only pseudo-automatic. In the following we present its basic features, several internal details and some results of using the prototype.

[1]https://goo.gl/YRMYVI

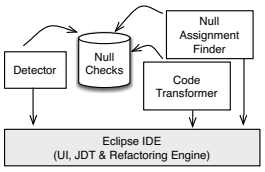

Fig. 1: NULLTERMINATOR Architecture

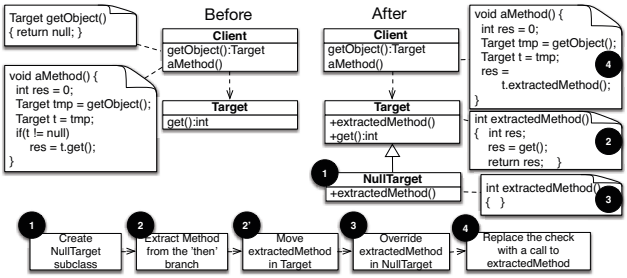

Fig. 2: A Transformation Example

II. ARCHITECTURE AND FEATURES

Figure 1 shows the basic modules of NULLTERMINATOR . They correspond to the main processing tasks of the tool and are described in the following.

Detector. When analyzing a project, we start by searching and recording information about the existing 'null' checks (e.g., the involved variable, the used relational operator, data about the true/false branches, etc.). This activity is performed by traversing the ASTs provided by ECLIPSE-JDT. During this traversal, we also filter out checks that the tool cannot remove (e.g., cases when a final or library class should be derived in order to introduce the *Null Object* pattern). We also eliminate the cases where the 'null' checked variable does not appear in one of the branches of the 'if' node, as in this case applying the pattern would just introduce unnecessary complexity to the code (i.e., the source code from the branches is probably not related to the class of the checked variable).

Code Transformer. The user can navigate through the identified 'null' checks, can browse the corresponding source code and, when considered necessary, she can trigger the execution

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.23

of the restructuring procedure. This module uses some of the "simpler" refactorings already available in ECLIPSE (e.g., Extract Method) and its concrete sequence of actions depends on the form of the 'null' check. Figure 2 describes an example.

In a first step, following the *Null Object* insertion approach from [1], the *NullTarget* subclass is created. Next, using *Extract and Move Method* refactorings, the code from the 'then' branch of the check is extracted into a new method and moved into the *Target* class. In a third step, the method is overridden in the *NullTarget* class and, finally, the entire check is replaced by a polymorphic call to the newly created operation. We mention here that the appearance of the *extractedMethod* is not specific for the introduction of the *Null Object* pattern (e.g., the same effect could be obtained by overriding the *get* method in *NullTarget*). With the risk of polluting the interface of the *Target* class, we adhere to this approach due to the advantage of generalizing the transformation.

The previous example also emphasizes the pseudo-automatic nature of our tool: it will be the developer responsibility to implement the overriding method from the *NullTarget* class. However, manually implementing this method is not always necessary (e.g., if, in our example, the extracted method does not have to return a value).

To handle other 'null' check forms, other specific actions might be required. For instance, if the check had an 'else' block, it would have been extracted in another method and moved into the *NullTarget* class. Moreover, if the methods extracted from the 'then' and 'else' branches have different signatures, the tool will have to unify them.

Null Finder. In order to complete the introduction of the pattern, all 'null' values assigned to the reference from the removed 'null' check must be replaced with an instance of the *Null Object* class. In the previous example, the *t* reference receives the 'null' value from the *getObject* method. Thus, the method will have to return a reference to a *NullTarget* object. Because these modifications are delicate, we do not apply them automatically. However, a search algorithm is available to the user in order to find and inspect the probable relevant 'null' values that have to be replaced.

The searching procedure is iterative, object- and flow-insensitive, inter-procedural and starts with a 'null' checked reference (i.e., *t* in our example). The AST of the method containing the check is iteratively traversed in order to detect all the direct or indirect assignments to the *t* variable found in the boundaries of the current method. If an assignment having 'null' in the right-hand side is reached, it is reported to the user. In our example, it will be determined that *t* receives the value from *tmp* which receives the value from *getObject()*.

When the 'null' value may come from another method, the previous procedure is applied starting from the relevant variable in that method. In our example, we continue by searching assignments to a special *retVar* reference in the *getObject* method. The assignment corresponding to the *return null* statement will be found and reported to the user.

Similarly, many other cases are considered. When the value of a formal parameter may flow to a 'null' check, the actual parameters from all call sites are passed through the same searching procedure. When a field is involved, all the methods accessing the field are searched (including constructors and initializing expressions). Thus, a method could have to be re-investigated when it accesses a field but, at the time of its previous analysis, it has not been determined yet that the field value may flow to the 'null' check. The procedure stops when no new indirect variables / relevant assignments are found.

The procedure has limitations. For instance, it is be possible to find 'null' value assignments that cannot actually reach the targeted 'null' check point. However, it is highly possible that such assignments will have to be also transformed in the context of some other 'null' check removal. As another limitation, 'null' values originating or traversing library code will not be detected also.

III. RESULTS AND CONCLUSIONS

For an initial evaluation of our prototype tool, we performed some tests on FindBugs and JDeodorant projects, finding 340, respectively 365 'null' checks candidates for applying the restructuring. We investigated 70 cases by trying to perform the refactoring and we achieved the results in Table I. Since the tool is aimed to be pseudo-automatic, we considered a case as being successful when, to the best of our understanding, the tool i) properly inserted the structure of the pattern and ii) it emphasized all the relevant 'null' assignments, even if some false-positives were also detected.

TABLE I: Results

Project	LOC	'null' checks found	Tried cases	Successful cases
JDeodorant	77,303	365	35	24
FindBugs	186,347	340	35	21

At this moment, the failed cases for the pattern structure insertion can be split in two categories: those caused by limitations of the Eclipse restructuring engine and others that need a refinement of the refactoring steps. The most common fails are because the extracted method would need to return multiple values or when the extracted method from the 'then' branch needs to return a different kind of value from the one from the 'else' branch. These problems could be solved by packing the values in an object. However, we did not implement this since in most cases it would complicate the code too much. Regarding the 'null' finding algorithm, the main problem is caused by 'null's coming from collections.

In conclusion our tool looks promising, helping in the introduction of the *Null Object* pattern in a relevant number of cases. For future versions we are aiming to address the mentioned limitations. The main focus will be on the improvement of the 'null' finding algorithm in order to detect places where 'null' values are added to a relevant collection or originating from a collection (i.e., retrieving the value for a key that is not available in a Java map). Furthermore, we plan to address potential challenges raised by the usage of exceptions and related 'null' checked variables.

REFERENCES

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

[2] B. Woolf, "Null Object," in *Pattern Languages of Program Design 3*, R. C. Martin, D. Riehle, and F. Buschmann, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, pp. 5–18.

[3] "Eclipse," http://www.eclipse.org/, accessed: 2016-05-30.

[4] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Type-Checking Bad Smells," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 329–331.

[5] M. A. G. Gaitani, V. E. Zafeiris, N. Diamantidis, and E. Giakoumakis, "Automated Refactoring to the Null Object Design Pattern," *Inf. Softw. Technol.*, vol. 59, no. C, pp. 33–52, Mar. 2015.

[6] S. Kimura, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Does Return Null Matter?" in *Proceedings of the Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, February 2014, pp. 244 – 253.