Enhancing Software Robustness by Detecting and Removing Exception Handling Smells: An Empirical Study

Chin-Yun Hsieh, You-Lun Chen, Zhen-Jie Liao

Department of Computer Science and Information Engineering,
National Taipei University of Technology, Taiwan
hsieh@csie.ntut.edu.tw, {t103598008, t103598012}@ntut.edu.tw

Abstract

We propose a systematic way to uncover and fix bugs through detecting smells associated with exception handling. First, code of software under improvement is scanned for exception handling smells by a static analysis tool. The smells are reviewed for confirming if they are bugs by writing failing tests. Finally, code that contains the smells is refactored until the failing test passes and the smells are removed. We have also conducted an empirical study to demonstrate the efficacy of the proposed approach. In the empirical study, an open source static analysis tool is applied to detect exception handling smells in an open source web application. The result shows that out of the 357 smells reported by the tool, 124 are confirmed to be bugs that could affect the robustness of the web application.

Keywords: Code smells, Robustness, Exception handling, Refactoring, Software testing

1 Introduction

Robustness [1-2] of software is the capability for software to continue to operate normally or degrade gracefully in the presence of faults. While tactics at both system and component levels are applicable to achieve robustness, exception handling is a critical link among them. For instance, in a system with shadow redundancy, the secondary kicks in when failure of the primary is detected or notified. The detection/notification often involves exception handling. As another instance, software that connects to the outside world often will need to tolerate connection faults [3-4]. The fault tolerance behavior depends on the software's capability to detect connection faults and intervene with proper handling strategies such as retrying or attempting an alternative connection.

For fault detection and handling [5-6] to work, code in exception handling [7] must be written correctly. However, previous empirical research has shown this to be a challenging task for programmers, especially for novices. Code that fails to meet the challenge is often infested with smells, e.g., empty catch blocks that ignore exceptions caught, catch blocks that cosmetically handle exception without resolving the fault, improper placement of resource cleanup that leads to resource leak, and so on [8-9]. Thus, the presence of smells around exception handling constructs of try, catch and finally blocks is often a good indicator of the presence of bugs. Furthermore, bugs lurking behind incorrect code for exception handling are not easily uncovered by testing since the exception-handling behavior of a program is often the least understood and poorly tested part [10].

In this paper, we present a systematic way to improve the robustness of software through the detection and removal of smells in code for exception handling. In this approach, exception handling smells are first detected through tool of static analysis [11-12]. The results are reviewed by developers. A detected smell is confirmed to be a bug if a failing test can be written to reproduce the manifestation of the bug. In confirming with a failing test, an exception of a specific type is injected at the designated points of the program to set off the program's exception handling behavior, and the failing condition is captured in the test. The developers then refactor exception handling code to remove both the bug and the smell, the success of which is marked by the passing of the previously failing test [13]. The removal of smell is further confirmed by applying static analysis yet again to show the absence of smells. The process is repeated until no smells are detected as illustrated in Figure 1.

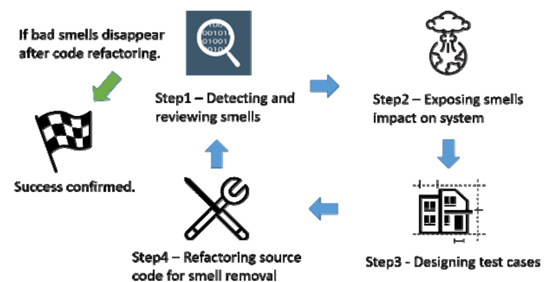

Figure 1. Steps of the smells removal process

*Corresponding Author: Chin-Yun Hsieh; E-mail: hsieh@csie.ntut.edu.tw
DOI: 10.3966/160792642018091905031

The efficacy of the method is demonstrated with its application for improving the robustness of the core of ezScrum, an open source web application for Scrum process support [14], which consists of more than thirty thousand lines of code in Java. The results confirm the conjecture that bugs tend to accompany exception handling code that has smells. Using Robusta, an open source static analysis tool for detecting exception handling smells [15], 357 exception handling smells are reported. Within the 357 smells, 124 bugs are found and confirmed. We demonstrate that the improvement process indeed helps to expose many sublime bugs related to exception handling that previously eschewed the developers of ezScrum even though a substantial number of unit tests, integration tests, and acceptance tests have been accumulated and run in ezScrum's daily build.

The rest of this paper is organized as follows. Section 2 gives an overview of related work. Section 3 presents the results of applying the proposed method to improve the robustness of ezScrum and elaborates on two bugs that are found hiding behind code smells. Section 4 demonstrates how to write tests for confirming and exposing bugs with a keyword-driven acceptance test written in Robot Framework [16] and with an integration test written in Junit [17], respectively. Section 5 shows how the bugs and smells are removed by refactoring. Finally, Section 6 offers a brief conclusion.

2 Backgrounds

In this section, the related information is given as background, including exception handling smells in Java, ezScrum, and AspectJ.

2.1 Exception Handling Smells in Java

Table 1 shows the names, definitions and effects of the exception handling smells [8-9] that Robusta – the static analysis tool used in this study – is able to detect in a Java program.

Table 1. Exception handling smells detectable by Robusta

Smell	Definition	Effect
Empty Catch Block	Nothing is done after catching an exception.	A potential fault is falsely ignored.
Dummy Handler	An exception is recorded or logged only.	A potential fault may not be resolved as it should be.
Nested Try Statement	A try-block is contained in the try, catch, or finally block of another try statement.	Complicates program logic and the debugging task.
Unprotected Main Program	A main function that has no enclosing try block.	System may be terminated even for a minor error.
Careless Cleanup	A resource may be prevented from being closed by a raised exception.	May lead to resource leak.
Exception Thrown From Finally Block	An exception is raised in the finally block.	May overwrite an exception thrown previously in the try block or any of the catch blocks.

2.2 ezScrum

ezScrum is a web application for supporting the agile process Scrum and has been on SourceForge since March, 2010 [14]. ezScrum is used by developers to manage user stories, keep track of development activities, and generate reports and analytics. ezScrum allows developers from different place to work together on the same Scrum team. To date, more than ten thousand copies have been downloaded.

The development of ezScrum started 9 years ago with a Scrum team consisting of 4-6 graduate student developers each year ever since. The most recent release has a size of 36,376 lines of code for the core alone. The code base is covered by 1,010 unit/integration test cases with a coverage rate of up to 69%. There are also 90 acceptance test cases. All of the tests are executed on a continuous integration system running Jenkins in a daily build. So far, 116 bugs have been fixed, which are either reported by users or found by the developers. It is expected that ezScrum will keep evolving for some time to further enrich the functionality, improve the performance, and facilitate its use.

ezScrum is chosen as the subject of this study both because we are familiar with its design and because ezScrum has attracted a number of users around the world. Improving its robustness will make a good contribution to the open source community.

2.3 AspectJ

AspectJ [18] is an Aspect-Oriented Programming (or AOP for short) [19] extension to the Java language. It allows the developer to inject new statements to a Java program in runtime without changing the source code. In this study, aspect functions are implemented to throw an exception at a designated point in the code of ezScrum so that the effect of a smell related to the exception can be exposed.

Here we give a simple aspect code to elaborate a little more, as shown in Figure 2. There are some keywords used in AspectJ to help developers achieve their goals, such as pointcut, call, target, withincode and around, the usage of each of these keywords will be explained based on the code in Figure 2. At line 10, developers use pointcut and call to decide the target method. It means that if the write function is invoked, aspect code will be executed successively. At line 11, target is used to distinguish a certain class of objects

within the source code. However, in this paper, we utilize it for another purpose. That is, we would need the WritableWorkbook object to preserve the original feature of the write function. With target, we can invoke write function by using WritableWorkbook object when there is no need of throwing an exception, like line 18. At line 12, the withincode is used to limit the target of aspect injection. Semantically, this line of code means that only write function, which is invoked in write Data To Temp File function of Exprot Stories From Product Backlog Action object, will be injected. Finally, around is used to decide when the program will invoke aspect function. This keyword is special in that it will overwrite the original feature of the target method statement. Like line 14, it will overwrite the feature of write function from writing information to front-end to throwing exception. Therefore, it needs to cooperate with target to keep the original feature when there is no need to throw an exception. Once the setting of these keywords is correctly done, this aspect function will inject the exception to expose code smell impact on software.

```
 9 public aspect ExportStoriesFromProductBacklogActionAspect {
10     pointcut findWrite(WritableWorkbook workbook) : call(void WritableWorkbook.write())
11     && target(workbook)
12     && withincode(void ExportStoriesFromProductBacklogAction.writeDataToTempFile(..));
13
14     void around(WritableWorkbook workbook) throws IOException : findWrite(workbook){
15         if (AspectJSwitch.getInstance().isSwitchOn("ExportStoriesFromProductBacklogAction")) {
16             throw new IOException();
17         } else {
18             workbook.write();
19         }
20     }
21 }
```

Figure 2. A sample of aspect code

3 Smells Detection and Analysis

The result of applying Robusta to scan ezScrum for exception handling smells is shown in Table 2. It is interesting to note that Dummy Handler and Careless Cleanup together account for up to 96 percent of total smells instances detected. In this section, two types of examples are elaborated, respectively.

Table 2. Exception handling smells detected in ezScrum

	Number of smell instances	Percentage
Empty Catch Block	4	1.12%
Dummy Handler	264	73.95%
Nested Try Statement	6	1.68%
Unprotected Main Program	2	0.56%
Careless Cleanup	79	22.13%
Exception Thrown From Finally Block	2	0.56%
Total	357	100%

3.1 Verification of the Smells

The following presents the results in verifying all of the detected smells regarding their impacts to ezScrum. The verification involves three steps. First, we refer to the smell report generated by Robusta to find the location of a smell. Second, expose the impact of the smell by throwing exception. Third, compare ezScrum's behavior which encounters exception with its normal behavior.

A smell is considered as a bug if, after we expose the impact of smell, it will make ezScrum function improperly without showing any error message. As such, users or developers would not be aware of the issue. The result of the verification is given in Table3.

As shown in Table 3, some of the smells are not considered as real bugs after the verification process. These cases can be divided into three groups based on their characteristics. The first group will not influence ezScrum because the raised exception is simply swallowed somewhere in propagation. The second group is not considered as real bugs because, even though they do make ezScrum function improperly, the users or developers will be informed about the situation to prevent it from getting worse. The third group is special in nature as a case of Dummy Handler shown in Figure 3. It perfectly fits into the definition of a smell, i.e., nothing else is done in the catch block except printing a message to the console (line 46). However, the present case is not considered as a smell because there is no other way to handle the issue other than printing a message to the console and ignoring the exception. In general, exceptions that are thrown from closing resource function, such a strategy is acceptable.

Table 3. Exception handling smells and bugs confirmed

Exception handling smell	Number of smell instances (A)	Number of bugs confirmed (B)	B/A (%)
Empty Catch Block	4	1	25
Dummy Handler	264	116	43.9
Nested Try Statement	6	0	0
Unprotected Main Program	2	1	50
Careless Cleanup	79	5	6.3
Exception Thrown From Finally Block	2	1	50
Summary	357	124	34.7

```
41 protected void closeResultSet(ResultSet result) {
42     if (result != null) {
43         try {
44             result.close();
45         } catch (SQLException e) {
46             e.printStackTrace();
47         }
48     }
49 }
```

Figure 3. A Dummy Handler that is acceptable

3.2 A typical Example of Dummy Handler

In Scrum, an unplanned item is a user story or a task that is not deliberated in sprint planning, but is accepted by both the product owner and the Scrum team of its urgency to justify its completion in the current sprint. Figure 4 shows a code snippet of method ShowEditUnplanItemAction for providing the attributes of a selected unplanned item to the front-end. The code snippet is marked to have a Dummy Handler smell at line 54. According to Java API documentation, an IOException may be raised at line 51 when the getWriter() function is invoked on the response object. The IOException will be caught by the catch-block at line 53 and handled at line 54, which simply logs the exception. In effect, this will cause the exception to be ignored. Thus, it is reasonable to conjecture that a bug is caused by a Dummy Handler smell.

```
49 try {
50     response.setContentType("text/xml; charset=utf-8");
51     response.getWriter().write(result.toString());
52     response.getWriter().close();
53 } catch (IOException e) {
54     e.printStackTrace();
55 }
```

Figure 4. Code segment of execute method in ShowEditUnplanItemAction marked to have a smell.

The execution of the use case that confirms the conjectured bug associated with the detected Dummy Handler smell is as follows. In ezScrum, the user can add a task as an unplanned item, which was not scheduled during sprint planning. Once the unplanned item is added, the user can modify it as necessary, e.g., to update the remaining hours until completion. To do this, the user first selects the desired unplanned item from the Unplanned Page and then clicks the Edit Unplanned Item button. After that, the front-end will send a request to the back-end to retrieve the attributes of the selected unplanned item. When the request is received by the back-end, method Show Edit Unplan Item Action will be executed and an Edit Unplanned Item window containing the retrieved attributes pops up, as shown in Figure 5.

When submitting the modified unplanned item, the front-end sends a request with identity number #1, which is shown on Edit Unplanned Item window's title, to the back-end. At this point, the original unplanned item stored in database will be updated. Once the update is done, the Edit Unplanned Item window is closed and the modified unplanned item is shown in the Unplanned Item List, as shown in Figure 6.

However, if an IOException is thrown when invoking response.getWriter(), the call to function write(result.toString()) in line 51 will not be invoked. As a result, the front-end receives a response with empty content after the user clicks Edit Unplanned Item button, and ezScrum pops up a window without attributes as a response, where the identity number on window's title does not show. Additionally, the Submit button is disabled because the required field is empty, as shown in Figure 7.

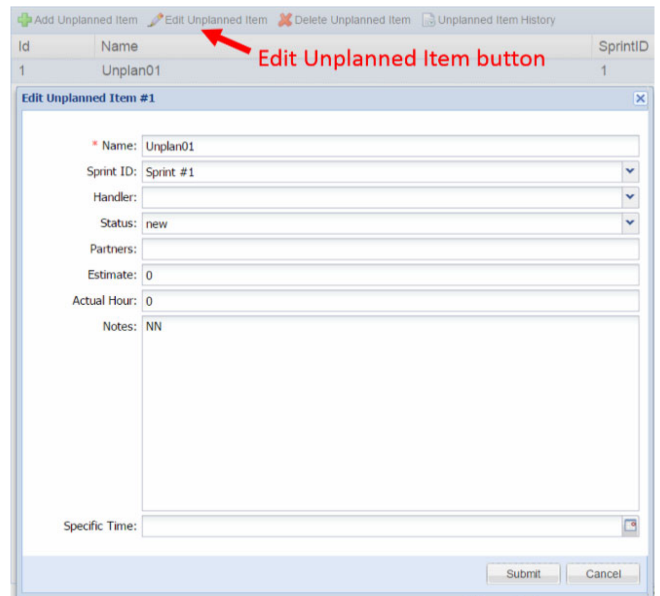

Figure 5. The Edit Unplanned Item window containing an unplanned item's attributes

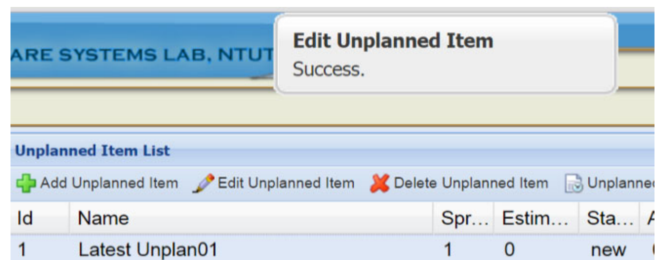

Figure 6. Unplanned Item List showing an unplanned item.

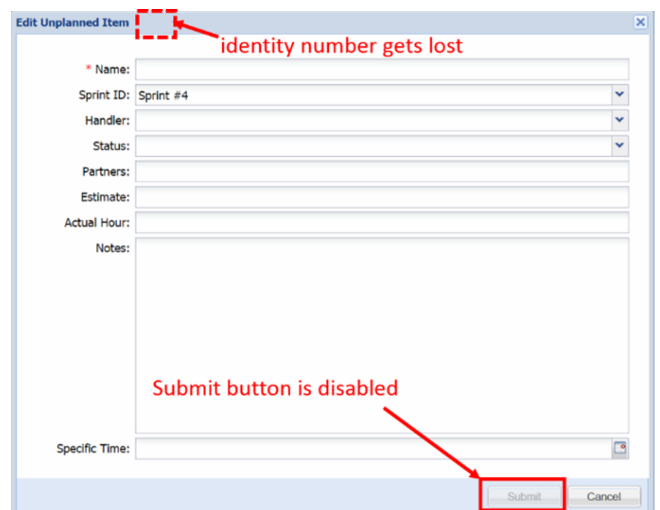

Figure 7. The Edit Unplanned Item window missing some required information

Even if the user inputs the required information and submits the modified unplanned item, the front-end

will send a request without identity number. Thus, the modifying unplanned item request will be rejected by the back-end because it loses its identity number, as shown in Figure 8. This is certainly a bug.

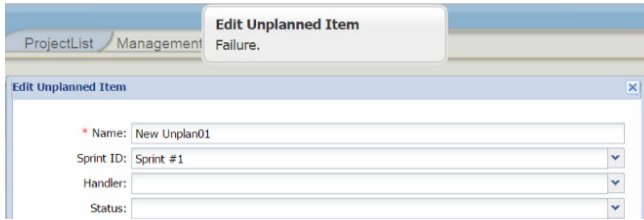

Figure 8. A request for modifying unplanned item that fails due to the loss of identity number

3.3 A Typical Example of Careless Cleanup

For the code snippet in Figure 9, an IOException may be raised when function write() is invoked on object workbook (line 50). The exception prevents workbook.close() at line 52 from being invoked. The result is that the file resource ezScrumExcel, which is created by the File object in line 38, will not be released as planned. This is exactly a Careless Cleanup as defined. A use case in ezScrum related to this is presented below.

```
37 // set the path of the temporary file
38 File mTempFile = File.createTempFile("ezScrumExcel", Long.toString(System.nanoTime()));
39 String mPath = mTempFile.getAbsolutePath();
40
41 try {
42     // create Excel
43     WritableWorkbook workbook = Workbook.createWorkbook(new File(mPath));
44     WritableSheet sheet = workbook.createSheet("BACKLOG", 0);
45     // delicate to excel handler
46     ExcelHandler handler = new ExcelHandler(mProject.getId(), sheet);
47     handler.save(mStories);
48
49     // write data to WritableWorkbook
50     workbook.write();
51     // release WritableWorkbook
52     workbook.close();
53 } catch (Exception e) {
54     e.printStackTrace();
55 }
56 flushFile(mTempFile);
57 mTempFile.delete();
```

Figure 9. Code snippet of method getStreamInfo in ExportStoriesFromProductBacklogAction.

In the Product Backlog Page of ezScrum, the user can export all user stories in product backlog to a Microsoft Excel file. It works in this way: a request will be sent to the back-end when the Export Story button (Figure 10) at the front-end is clicked; the getStreamInfo function will be invoked in Export Stories From Product Backlog Action when the request is received by the back-end; and the front-end will receive an excel file which contains all user stories in product backlog when getStreamInfo finishes executing (as shown in Figure 11).

Now consider what happens if an IOException is raised when workbook.write() is invoked at line 50. Apparently, the workbook.close() function at line 52 will not be executed. Consequently, the ezScrumExcel file created at line 38 is not released, as shown in Figure 12.

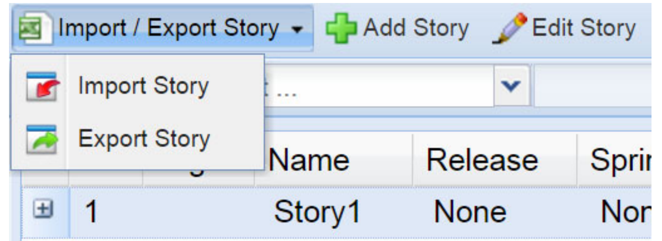

Figure 10. Export Story button in Product Backlog page

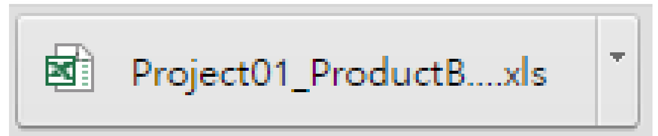

Figure 11. Excel file downloaded after export

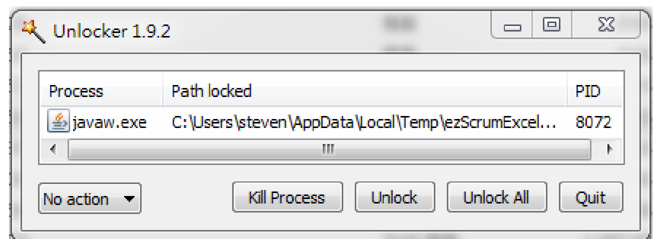

Figure 12. ezScrumExcel file shown to be unreleased (shown with Unlocker [20].)

As a result, the ezScrumExcel file cannot be removed by executing mTempFile.delete() function at line 57. Note that an ezScrumExcel file will be generated every time the user clicks the Export Story button. The accumulated size of the non-deleted ezScrumExcel files will certainly become a waste of storage space. To prevent this from happening, the workbook.close() function should be moved to the finally-block.

4 Writing Tests for Confirming and Exposing Bugs

To expose each of the two bugs behind the smells of Section 3, a test case is written based on the respective use case. An aspect function is used to throw an exception at a designated point for both test cases. Note that both the type of the exception and the location where it occurs are readily found in Robusta. These test cases will also be used in Section 5 to verify the correctness of ezScrum's behavior after smell removal.

4.1 A Test Case for Testing the Typical Example of Dummy Handler

Figure 13 shows the aspect function used to throw an IOException (using the around option) when the response.getWriter() at line 51 of Figure 4 is called.

```
12 public aspect ShowEditUnplanItemActionAspect {
13     pointcut findGetWriter(HttpServletResponse response) :
14         call (PrintWriter HttpServletResponse.getWriter())
15         && target(response)
16         && withincode(ActionForward ShowEditUnplanItemAction.execute(..));
17
18     PrintWriter around(HttpServletResponse response) throws IOException : findGetWriter(response){
19         if (AspectJSwitch.getInstance().isSwitchOn("ShowEditUnplanItemAction")) {
20             throw new IOException();
21         } else {
22             return response.getWriter();
23         }
24     }
25 }
```

Figure 13. An aspect function for throwing an IOException at the designated point

The test case, written in Robot Framework, is shown in Figure 14. As described previously, if an IOException is raised in executing response. Get Writer(), an error message needs to be shown instead of the Edit Unplanned Item window. Accordingly, the test case is designed to check the correctness that, after the Edit Unplanned Item button is clicked, an error messages box instead of the Edit Unplanned Item window is popped up. The error messages are "Server Error" and "Sorry, fail due to internal server error."

1	# Choose unplan		
2	Select Unplan	1	Unplan01
3	# Click Edit Unplanned Item		
4	Click Element	xpath=//button[.='Edit Unplanned Item']	
5	# Verify no Edit Unplaned Item window pop-up		
6	${EditUnplanItemWindow}=	Set Variable	//span[contains(text(), "Edit Unplanned Item")]
7	Wait Until Element Is Not Visible	${EditUnplanItemWindow}	
8	# Verify error message		
9	Wait Until Page Contains	Server Error	
10	Wait Until Page Contains	Sorry, fail due to internal server error	

Figure 14. A Robot Framework test case for testing the typical example of Dummy Handler

4.2 A Test Case for Testing the Typical Example of Careless Cleanup

Figure 15 shows the aspect function used to throw an instance of IOException when workbook.write() at line 50 of Figure 9 is executed.

```
 9 public aspect ExportStoriesFromProductBacklogActionAspect {
10     pointcut findWrite(WritableWorkbook workbook) : call(void WritableWorkbook.write())
11     && target(workbook)
12     && withincode(void ExportStoriesFromProductBacklogAction.writeDataToTempFile(..));
13
14     void around(WritableWorkbook workbook) throws IOException : findWrite(workbook){
15         if (AspectJSwitch.getInstance().isSwitchOn("ExportStoriesFromProductBacklogAction")) {
16             throw new IOException();
17         } else {
18             workbook.write();
19         }
20     }
21 }
```

Figure 15. An aspect function for throwing an IOException at the designated point

The test case, which is written in JUnit, is shown in Figure 16. As described previously, the ezScrumExcel file must be removed no matter if an IOException is raised or not when workbook.write() is executed. Accordingly, the test case is design to check that, after the getStreamInfo method in Export Stories From Product Backlog Action has finished executing, ezScrum Excel file is no longer in use and is removed from ezScrum.

```
75 public void testExportStoriesFromProductBacklogAction() {
76     // Turn AspectJ Switch on
77     AspectJSwitch.getInstance().turnOnByActionName(mActionName);
78     // invoke ExportStoriesFromProductBacklogAction
79     actionPerform();
80     File ezScrumExcel = getEzScrumExcelTempFile();
81     // if ezScrumExcel does not exist,
82     // getEzScrumExcelTempFile() will return null.
83     assertNull(ezScrumExcel);
84     // Turn AspectJ Switch off
85     AspectJSwitch.getInstance().turnOff();
86 }
```

Figure 16.A JUnit test case for testing the typical example of Careless Cleanup

5 Removing Smells by Refactoring and Confirming Their Success

This section presents the refactoring for removing the smells described in Section 3. In addition to passing the previously failing test, the removal is further confirmed by applying static analysis yet again, which must show the absence of the detected smell around the code.

5.1 Refactoring for Removing the Typical Example of Dummy Handler

Figure 17 shows the new version of Show Edit Unplan Item Action. It differs from the original version in that the aforementioned IOException is rethrown (line 55) after the exception message is recorded and an error message is displayed at the front-end. This enables the front-end to know that something has gone wrong in the back-end.

```
49 try {
50     response.setContentType("text/xml; charset=utf-8");
51     response.getWriter().write(result.toString());
52     response.getWriter().close();
53 } catch (IOException e) {
54     e.printStackTrace();
55     throw e;
56 }
```

Figure 17. A new ShowEditUnplanItemAction with the Dummy Handler smell removed

In ezScrum, when an unhandled exception is raised in ShowEditUnplanItemAction at the back-end, a status named "internal server error" is returned to the front-end. As shown in Figure 18, the front-end code sends the modifying unplanned item request. When the back-end receives the request, Show Edit Unplan Item Action will be invoked by the program. If the program executes Show Edit Unplan Item Action correctly, the back-end will return a response with a status named "OK", so that the success function (line 212) will be invoked. Then, the Edit Unplanned Item window will pop up. On the other hand, if an exception is raised during execution, the back-end will return a response with a status named "internal server error", hence invoking the failure function (line 216). Then, an error message is shown on the message box by invoking onLoadFailure function (line 217). Consequently, a message box with error messages is displayed, as

shown in Figure 19.

```
209 Ext.Ajax.request({
210     url:this.loadUrl,
211     params : {issueID : this.issueId},
212     success:function(response){
213         window.show();
214         obj.onLoadSuccess(response);
215     },
216     failure:function(response){
217         obj.onLoadFailure(response);
218     }
219 });
```

Figure 18. The front-end code designed to trigger ShowEditUnplanItemAction

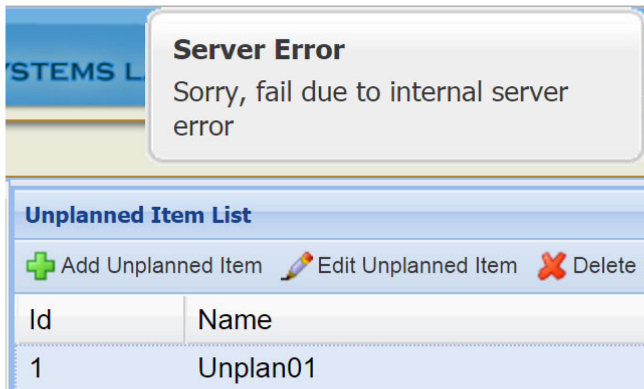

Figure 19. An error message is displayed as expected after the smell is removed by code refactoring

The revised code version has passed the previously failing test as shown in Figure 20. The removal is also confirmed by Robusta – the absence of a smell sign on the left margin of the code (as shown in Figure 17).

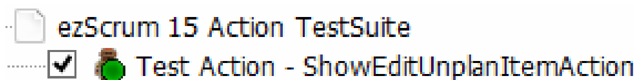

Figure 20. The failing test is passed after removing the Dummy Handler smell

5.2 Refactoring for Removing the Typical Example of Careless Cleanup

The Careless Cleanup in the code snippet in Figure 9 can be removed by moving the workbook.close() function to the finally block, as shown in Figure 21. By doing so, it is assured that the ezScrumExcel file will be released no matter if an IOException is thrown or not when workbook.write() at line 50 is executed. In other words, the ezScrumExcel file will always be removed from ezScrum after the execution of the get Stream Info function in Export Stories From Product Backlog Action.

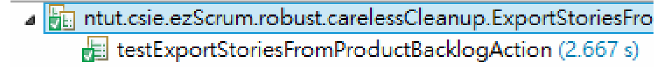

Figure 21. Test results showing the removal of the Careless Cleanup

The revised code successfully passes the previously failing test and the ezScrumExcel file is deleted as expected, as shown in Figure 21. The removal is also confirmed by Robusta (Figure 22).

```
40 WritableWorkbook workbook = null;
41 try {
42     // create Excel
43     workbook = Workbook.createWorkbook(new File(mPath));
44     WritableSheet sheet = workbook.createSheet("BACKLOG", 0);
45     // delicate to excel handler
46     ExcelHandler handler = new ExcelHandler(mProject.getId(), sheet);
47     handler.save(mStories);
48
49     // write data to WritableWorkbook
50     workbook.write();
51 } catch (Exception e) {
52     e.printStackTrace();
53 } finally {
54     if (workbook != null) {
55         // release WritableWorkbook
56         workbook.close();
57     }
58 }
59 flushFile(mTempFile);
60 mTempFile.delete();
```

Figure 22. A new version of the code snippet in Figure 9 with Careless Cleanup smell removed

Retrospectively, the detailed process of smell confirmation and removal presented in Sections 3, 4 and 5 is illustrated in the flowchart of Figure 23.

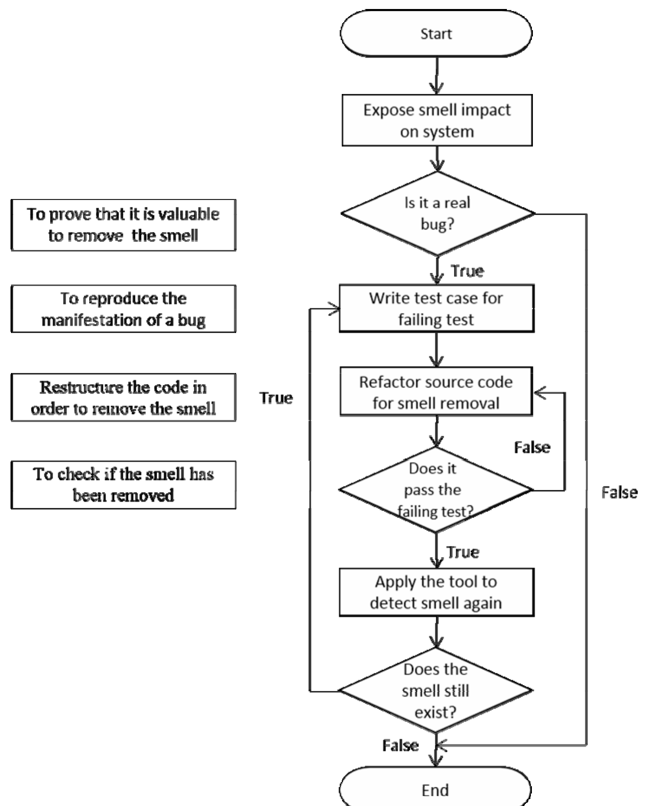

Figure 23. Flowchart of smell confirmation and removal process

6 Conclusion

This paper shows that bugs that affect robustness of software can often be found near code infested with exception handling smells. A systematic way for enhancing the robustness by detecting and removing exception handling smells is proposed. A static analysis tool is first applied to detect smells in a program. The detected smells are reviewed by developers. A failing test is then used to confirm if a smell is really a bug. Source code associated with a confirmed bug is refactored, required to pass the original failing test, and reconfirmed by static analysis. A good result is achieved in an empirical study employing the proposed approach.

Although the proposed approach can help enhance software robustness, the code review and failing tests writing are time-consuming if the number of detected smells is large. If there is a tool, which can generate failing tests for verifying if a smell is a bug, the effort needed will be greatly reduced. We leave this as our future work.

Acknowledgements

The research was sponsored in part by the Ministry of Science and Technology under the grant MOST104-2221-E-027-007.

References

[1] J. C. Fernandez, L. Mounier, C. Pachon, A Model-Based Approach for Robustness Testing, *17th IFIP TC6/WG 6.1 International Conference*, Montreal, Canada, 2005, pp. 333-348.

[2] J. W. Baker, M. Schubert, M. H Faber, *On the Assessment of Robustness*, Structural Safety, Vol. 30, No. 3, pp. 253-267, May, 2008.

[3] P. A. Lee, T. Anderson, *Fault Tolerance: Principles and Practice*, Springer-Verlag Wien, 1990.

[4] K. Whisnant, Z. Kalbarczyk, R. K. Iyer, A Foundation for Adaptive Fault Tolerance in Software, *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, Oxford, UK, 2003, pp. 252-260.

[5] W. Jiang, D. Ma, Y. Zhao, Strategy-based Fault Handling Mechanism for Composite Service, *2013 Third International Conference on Intelligent System Design and Engineering Applications*, Hong Kong, China, 2013, pp. 1297-1301.

[6] M. Steinegger, A. Zoitl, M. Fein, G. Schitter, Design Patterns for Separating Fault Handling from Control Code in Discrete Manufacturing Systems, *IECON 2013- 39th Annual Conference of the IEEE Industrial Electronics Society*, Vienna, Austria, 2013, pp. 4368-373.

[7] S. Sinha, M. J. Harrold, Analysis and Testing of Programs with Exception Handling Constructs, *IEEE Transactions on Software Engineering*, Vol. 26, No. 9, pp. 849-871, September, 2000.

[8] C. T. Chen, Y. C. Cheng, C. Y. Hsieh, I. L. Wu, Exception Handling Refactorings: Directed by Goals and Driven by Bug Fixing, *Journal of Systems and Software*, Vol. 82, No. 2, pp. 333-345, February, 2009.

[9] C. Y. Hsieh, H. H. Chen, Y. F. Chen, On the Evaluation of Performance and Cost Saving of Exception Handling Smells Detection through Static Code Analysis, *Journal of Systems and Software*, Vol. 115, May, 2016.

[10] S. Sinha, M. J. Harrold, Criteria for Testing Exception-handling Constructs in Java Programs, *Proceedings of the IEEE 15th International Conference on Software Maintenance*, Oxford, UK, 1999, pp. 265-274.

[11] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward, D. W. R. Marsh, *Industrial Perspective on Static Analysis, Software Engineering Journal*, Vol. 10, No. 2, pp. 69-75, May, 1995.

[12] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh, Using Static Analysis to Find Bugs, *IEEE Software*, Vol. 25, No. 5, pp. 22-29, September, 2008.

[13] J. Shore, Fail Fast, *IEEE Software*, Vol. 21, No. 5, pp. 21-25, September, 2004.

[14] ezScrum at SourceForge, https://sourceforge.net/projects/ezscrum/.

[15] Robusta at Eclipse Marketplace, https://marketplace.eclipse.org/content/robusta-eclips e-plugin.

[16] Robot Framework, http://robotframework.org/.

[17] Junit, http://junit.org/junit4/.

[18] AspectJ at Eclipse, https://eclipse.org/aspectj/.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, J. Irwin, Aspect-oriented Programming, *Proceedings of the 11th European Conference on Object-Oriented Programming*, Jyväskylä, Finland, 1997, pp. 220-242.

[20] Unlocker 1.9.2, http://filehippo.com/download_unlocker/.

Biographies

Chin-Yun Hsieh is a Professor at the Department of Computer Science and Information Engineering of the National Taipei University of Technology, Taiwan. He received his MS and Ph.D. degrees from the University of Mississippi and the University of Oklahoma, respectively, both in Computer Science. His research interests include pattern languages and software testing.

You-Lun Chen was born in Taoyuan, Taiwan, in 1991. He received the master degree in computer science and information engineering from the National Taipei University of Technology, Taipei, Taiwan, in 2016. His current research interests include

AspectJ, eclipse plugin developing and code quality. He is presently working as a quality assurance engineer for a software company.

Zhen-Jie Liao was born in Taipei, Taiwan, in 1992. He received the master degree in computer science and information engineering from the National Taipei University of Technology, Taipei, Taiwan, in 2016. His current research interests include exception handling, testing automation developing and CI/CD. He is presently working as a quality assurance engineer for a private company.