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

謝金雲、陳友倫、廖振傑
國立台北科技大學資訊工程系
Chin-Yun Hsieh, You-Lun Chen and Zhen-Jie Liao
Department of Computer Science and Information Engineering
National Taipei University of Technology
Email: hsieh@csie.ntut.edu.tw, {t103598008, t103598012}@ntut.edu.tw

Abstract

We propose a systematic way to uncover bugs associated with exception handling. First, code of software under improvement is scanned for exception handling smells by static analysis. 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. The proposed method has been applied to uncover a number of bugs that could affect robustness of an open source web application.

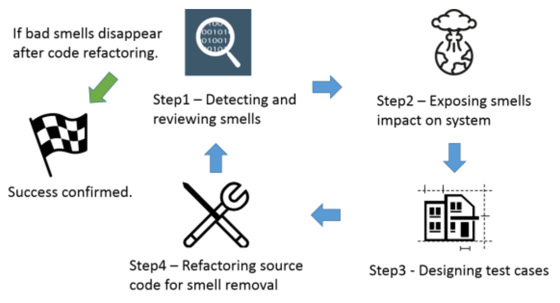
Keywords：code smells, robustness, exception handling, refactoring, aspect oriented programming, code review, software testing

1 Introduction

Robustness 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. 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 to work, code in exception handling must be written correctly. However, previous empirical research has shown this to be a challenging task for programmers, especially for novices [1]. 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 [2,3]. 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 [4].

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. The results are reviewed by developers. A detected smell is confirmed to be a bug if a failing test [5] 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. 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.


If bad smells disappear after code refactoring.
Step1 – Detecting and reviewing smells
Step2 – Exposing smells impact on system
Success confirmed.
Step4 – Refactoring source code for smell removal
Step3 - Designing test cases
Figure 1 Steps of the smells removal process.

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 [6], 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 smells. Using Robusta, an open source static analysis tool for detecting exception handling smells [7], 357 exception handling smells are reported. Within the 47 smells examined so far, 30 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 presents the 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 exposing bugs with a keyword-driven acceptance test written in Robot Framework [8] and with an integration test written in JUnit [9], respectively. Section 5 shows how the bugs and smells are removed by refactoring. Finally, Section 6 offers a brief conclusion.

2 Background

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

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.

2.1 Exception handling smells in Java

Table 1 shows the names, definitions and effects of the exception handling smells [2,3,10] that Robusta – the static analysis tool used in this study – is able to detect in a Java programs.

2.2 ezScrum

ezScrum is a web application for supporting the agile process Scrum and has been on SourceForge since March, 2010 [6]. 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 work together on the same Scrum team. To date, more than ten thousand copies has 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 36376 lines of code for the core alone. The code base is covered by 1010 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 [11] is an Aspect-Oriented Programming (or AOP for short) [12] 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.

3 Smells detection and analysis

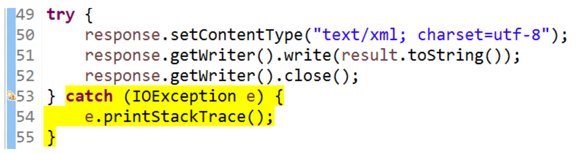
The result of applying Robusta to scan ezScrum for exception handling smells is shown in Table 2. It is interesting to note that Dummy Handler and Careless Cleanup together account for up to 96 percent of total smells instances detected. In this section, a typical example each of the two smells is elaborated.

Table 2 Exception handling smells detected in ezScrum.

	Number of smell instances	Percentage
Dummy Handler	264	73.94%
Careless Cleanup	79	22.12%
Nested Try Statement	6	1.68%
Empty Catch Block	4	1.12%
Unprotected Main Program	2	0.56%
Exception Thrown From Finally Block	2	0.56%
Total	357	100%

3.1 A typical example of Dummy Handler

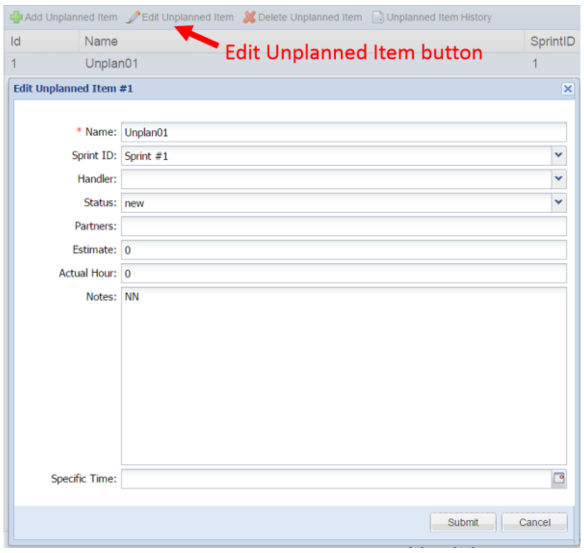
In Scrum, an *unplanned item* is a user story or a task that is not deliberated in sprint planning, but is accepted both by the product owner and the Scrum team of its urgency to justify its completion in the current sprint. Figure 2 shows a code snippet of method ShowEditUnplanItemAction for providing 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 2 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 follow. 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 till completion. To do this, the user first selects, from the in Unplanned Page, the desired unplanned item 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 ShowEditUnplanItemAction will be executed and an Edit Unplanned Item Window containing the retrieved attributes pops up, see Figure 3.



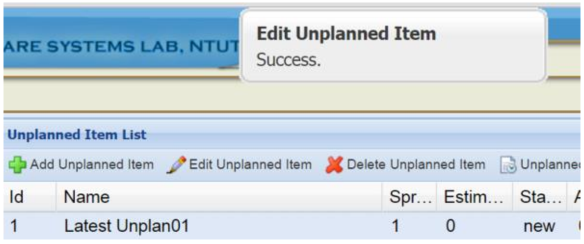
Id	Name	SprintID
1	Unplan01	1

Edit Unplanned Item #1

* Name: Unplan01
Sprint ID: Sprint #1
Handler:
Status: new
Partners:
Estimate: 0
Actual Hour: 0
Notes: NN
Specific Time:

Figure 3 The Edit Unplanned Item Window containing an unplanned item's attributes.

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, see Figure 4.



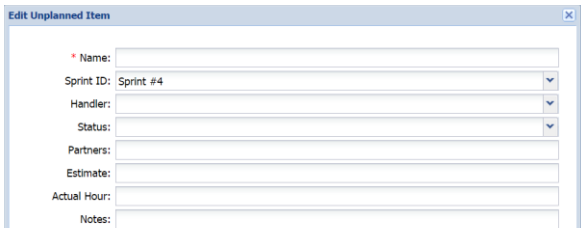
Edit Unplanned Item
Success.

Unplanned Item List

Id	Name	Spr...	Estim...	Sta...
1	Latest Unplan01	1	0	new

Figure 4 Unplanned Item List having an unplanned item.

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; see Figure 5. This is confirmed as a bug.



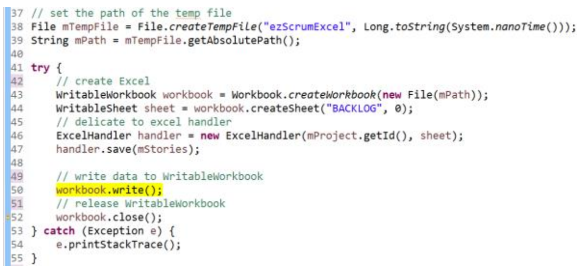
Edit Unplanned Item

* Name:
Sprint ID: Sprint #4
Handler:
Status:
Partners:
Estimate:
Actual Hour:
Notes:

Figure 5 The Edit Unplanned Item Window missing some required information.

3.2 A typical example of Careless Cleanup

For the code snippet in Figure 6, 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 temp 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 }
```

Figure 6 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 7) at the front-end is clicked; the getStreamInfo function will be invoked in ExportStoriesFromProductBacklogAction 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 (see Figure 8).

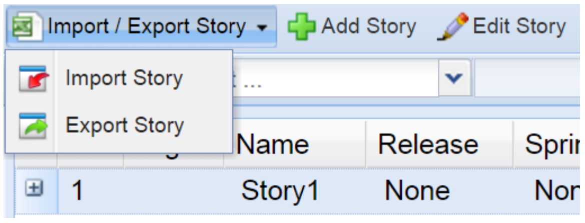


Figure 7 Export Story Button in Product Backlog Page

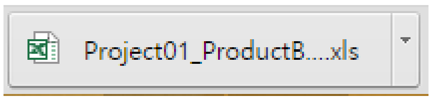
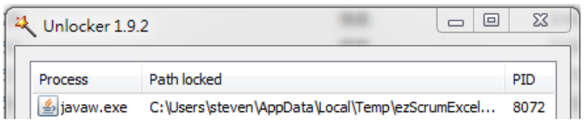


Figure 8 Excel file downloaded after export

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. As a result, the ezScrumExcel file created at line 38 is not released, see Figure 9. To prevent this from happening, the workbook.close() function should be moved to the finally-block.



Process	Path locked	PID
javaw.exe	C:\Users\steven\AppData\Local\Temp\ezScrumExcel...	8072

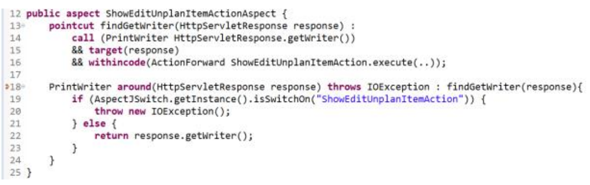
Figure 9 ezScrumExcel file shown to be unreleased (shown with Unlocker [13].)

4 Writing tests for confirming 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

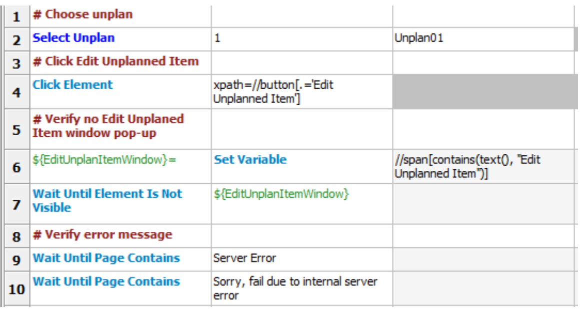
Shown in Figure 10 is the aspect function used to throw an IOException (using the around option) when the response.getWriter() at line 51 of Figure 2 is called.



```
public aspect ShowEditUnplanItemActionAspect {
    pointcut findGetWriter(HttpServletResponse response) :
        call (PrintWriter HttpServletResponse.getWriter())
        && target(response)
        && withincode(ActionForward ShowEditUnplanItemAction.execute(..));

    PrintWriter around(HttpServletResponse response) throws IOException : findGetWriter(response){
        if (AspectJSwitch.getInstance().isSwitchOn("ShowEditUnplanItemAction")) {
            throw new IOException();
        } else {
            return response.getWriter();
        }
    }
}
```

Figure 10 An aspect function for throwing an IOException at the designated point.



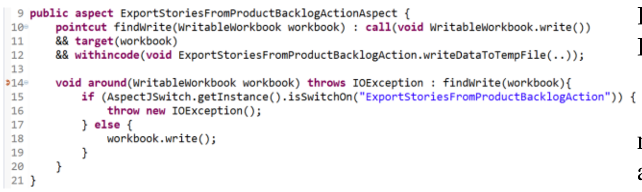
1	# Choose unplan		
2	Select Unplan	1	Unplan01
3	# Click Edit Unplanned Item		
4	Click Element	xpath=//button[.='Edit Unplanned Item']	
5	# Verify no Edit Unplaned Item window pop-up		
6	${EditUnplanItemWindow}=	Set Variable	//span[contains(text(), "Edit Unplanned Item")]
7	Wait Until Element Is Not 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 11 A Robot Framework test case for testing the typical example of Dummy Handler.

The test case, written in Robot Framework, is shown in Figure 11. As described previously, if an IOException is raised in executing response.getWriter(), 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."

4.2 A test case for testing the typical example of Careless Cleanup

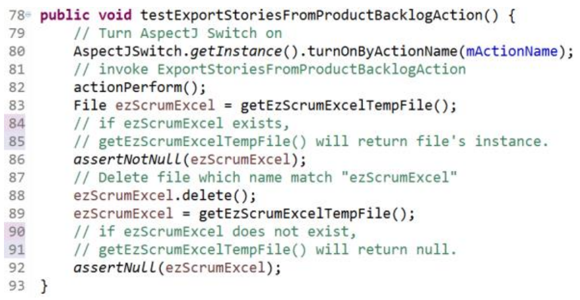
Shown in Figure 12 is the aspect function used to throw an instance of IOException when workbook.write() at line 50 of Figure 6 is executed.



```
public aspect ExportStoriesFromProductBacklogActionAspect {
    pointcut findWrite(WritableWorkbook workbook) : call(void WritableWorkbook.write())
    && target(workbook)
    && withincode(void ExportStoriesFromProductBacklogAction.writeDataToTempFile(..));

    void around(WritableWorkbook workbook) throws IOException : findWrite(workbook){
        if (AspectJSwitch.getInstance().isSwitchOn("ExportStoriesFromProductBacklogAction")) {
            throw new IOException();
        } else {
            workbook.write();
        }
    }
}
```

Figure 12 An aspect function for throwing an IOException at the designated point.



```
public void testExportStoriesFromProductBacklogAction() {
    // Turn AspectJ Switch on
    AspectJSwitch.getInstance().turnOnByActionName(mActionName);
    // invoke ExportStoriesFromProductBacklogAction
    actionPerform();
    File ezScrumExcel = getEzScrumExcelTempFile();
    // if ezScrumExcel exists,
    // getEzScrumExcelTempFile() will return file's instance.
    assertNotNull(ezScrumExcel);
    // Delete file which name match "ezScrumExcel"
    ezScrumExcel.delete();
    ezScrumExcel = getEzScrumExcelTempFile();
    // if ezScrumExcel does not exist,
    // getEzScrumExcelTempFile() will return null.
    assertNull(ezScrumExcel);
}
```

Figure 13 A JUnit test case for testing the typical example of Careless Cleanup.

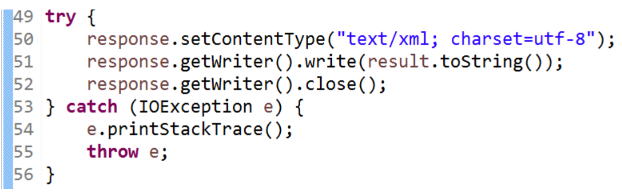
The test case, which is written in JUnit, is shown in Figure 13. As described previously, the ezScrumExcel file must be released by workbook no matter if an IOException is raised or not when workbook.write() is executed. Accordingly, the test case is design to check that, after the getStreamInfo method in ExportStoriesFromProductBacklogAction has finished executing, ezScrumExcel file is no longer in use.

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 pass the previously failing test, a 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

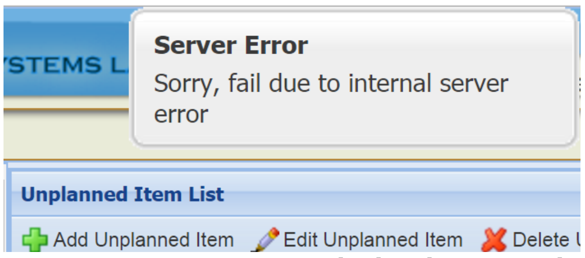
Shown in Figure 14 is the new version of ShowEditUnplanItemAction. It differs from the original version in that the aforementioned IOException is rethrown (line 55) after the exception message is recorded and an error message is displayed at the front-end. This enables the front-end to know that something 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 14 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 and a message box with error messages displayed, see Figure 15.

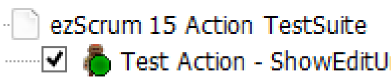


Server Error

Sorry, fail due to internal server error

Figure 15 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 16. The removal is also confirmed by Robusta – the absence smell sign on the margin of code – as shown in Figure 14.

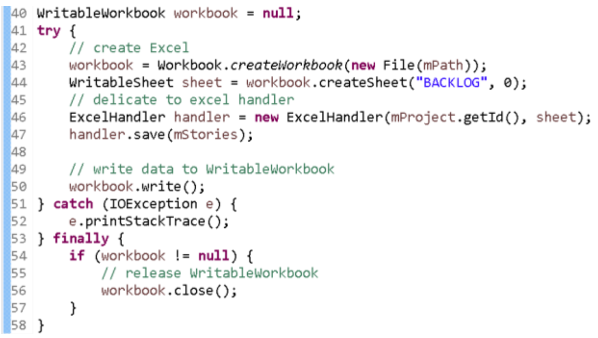


ezScrum 15 Action TestSuite
Test Action - ShowEditUnplanItemAction

Figure 16 Passing the failing test 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 6 can be removed by moving the workbook.close() function to the finally block, see Figure 17. By doing so, it is assured that the ezScrumExcel file will be released no matter if an IOException is thrown or not when workboo.write() at line 50 is executed.



```
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 (IOException e) {
52     e.printStackTrace();
53 } finally {
54     if (workbook != null) {
55         // release WritableWorkbook
56         workbook.close();
57     }
58 }
```

Figure 17 A new version of the code snippet in Figure 6 with Careless Cleanup smell removed.

The revised code successfully passes the previously failing test and the ezScrumExcel file is deleted as expected, see Figure 18. The removal is also confirmed by Robusta (Figure 17).

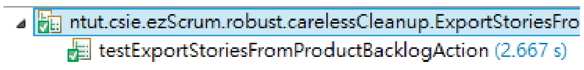

ntut.csie.ezScrum.robust.carelessCleanup.ExportStoriesFro
testExportStoriesFromProductBacklogAction (2.667 s)

Figure 18 Test result showing the removal of the Careless Cleanup.

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.

Acknowledgements

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

References

[1] Saurabh Sinha and Mary Jean Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol.26, no.9, pp.849-871, Sept. 2000.
[2] Chien-Tsun Chen, Yu Chin Cheng, Chin-Yun Hsieh and I-Lang Wu, “Exception Handling Refactorings: Directed by Goals and Driven by Bug Fixing,” *Journal of Systems and Software*, vol.82, no.2, pp.333-345, Feb. 2009 .
[3] Chin-Yun Hsieh, Hsin-Hung Chen, Yi-Fan Chen, “On the evaluation of performance and cost saving of exception handling smells detection through static code analysis,” submitted to the *Journal of Systems and Software*, May 2016.
[4] Saurabh Sinha and Mary Jean Harrold. "Criteria for testing exception-handling constructs in Java programs," *Proceedings of the IEEE 15th International Conference on Software Maintenance*, pp. 265-274, August 1999.
[5] Jennifer Shore, "Fail Fast," *IEEE Software*, vol. 21, no. 5, pp. 21-25, 2004.
[6] ezScrum at SourceForge, https://sourceforge.net/projects/ezscrum/
[7] Robusta at Eclipse Marketplace, https://marketplace.eclipse.org/content/robusta-eclipse-plugin
[8] Robot Framework, http://robotframework.org/
[9] Junit, http://junit.org/junit4/
[10] Chin-Yun Hsieh, Hsin-Hung Chen, Yi-Fan Chen, “On the evaluation of performance and cost saving of exception handling smells detection through static code analysis,” submitted to the *Journal of Systems and Software*, May 2016.
[11] Aspectj at Eclipse, https://eclipse.org/aspectj/
[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes,Jean-Marc Loingtier, John Irwin, “Aspect-oriented programming,” *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp.220–242, June 1997.
[13] Unlocker 1.9.2, http://filehippo.com/download_unlocker/