Agile Testing of Exceptional Behavior

Rafael Di Bernardo*, Ricardo Sales Jr†, Fernando Castor*, Roberta Coelho‡, Nélio Cacho‡, Sérgio Soares*

*Informatics Center, Federal University of Pernambuco
Recife, Brazil
†Informatics and Applied Mathematics Department, Federal University of Rio Grande do Norte
Natal, Brazil

Email: rafaelbernardo@gmail.com, castor@cin.ufpe.br, ricsales@gmail.com,
roberta@dimap.ufrn.br, neliocacho@ect.ufrn.br, scbs@cin.ufpe.br

Abstract—The lack of testing and a priori design of the exceptional behavior are causing many of the problems found in the use of exception handling. As a consequence, exceptions flow in unforeseen ways during the execution of a software system, having a negative impact on reliability. This paper presents an agile approach to test the exceptional behavior of a system. It supports developers in checking whether exceptions, at runtime, travel through the expected paths. It is agile because tests are written without the need for extra documentation and are, themselves, considered live documentation. We have evaluated our approach by applying it to different versions of two production quality Java open source applications (i.e., aTunes and JEdit). Using the proposed approach, we could find twelve bugs—eight of them previously unknown by the open source projects. In addition, from the viewpoint of automated tests as documentation artifacts, the proposed approach pointed out several differences between versions of the two target systems. We have implemented the proposed approach as an extension of the JUnit framework.

Keywords: exception handling, testing, exceptional behavior, aspect-oriented programming.

I. INTRODUCTION

Modern software systems are usually composed by a collection of distributed components that must deal with inputs from a variety of sources, execute in diverse environments, while addressing strict dependability requirements. Several techniques can be applied to address such dependability requirements. One of the most used techniques, which are embedded in mainstream programming languages, are the Exception Handling mechanisms [7]. These mechanisms help developers to build robust applications by separating exceptional behavior from the normal control flow [13]. They ensure system modularity in the presence of errors by offering abstractions for: (i) representing erroneous situations of system modules as exceptions, (ii) encapsulating exception handling activities into handlers, (iii) defining parts of the system modules as protected regions for handling exception occurrences, (iv) associating these protected regions with handlers, and (v) explicitly specifying exceptional interfaces of modules.

If, on the one hand, a large amount of code on modern application systems is dedicated to error detection and handling [23], [24], [9], on the other hand, dealing with manifestations of errors/exceptions at different stages of development (e.g., requirements elicitation, design tasks) has received little attention [17], [22]. Developers tend to give attention to the normal behavior of applications, not concentrating on the design of the exceptional activity [25]. They usually only deal with exception detection and handling in the implementation activities [7]. As a consequence, they do not properly use the exception handling mechanisms, compromising system dependability [22].

The exception handling constructs are usually fault prone [1], due to the lack of attention along software development disciplines. In other words, the code that was firstly designed to improve the system robustness becomes a source of faults. Approaches based on static analysis were proposed to discover such faults on the exception handling code. Such approaches are based on tools which discover the paths that exceptions take from signalers (i.e., elements that throw exceptions) to handlers (i.e., elements responsible for catching them) [14], [6], [16], [5], [29]. However, due to the limitations inherent to static analysis approaches combined with the characteristics of modern languages (such as inheritance, polymorphism and virtual calls such) such approaches usually report many false positives [28]. Hence, manual steps are usually required to verify whether the detected exception paths can actually happen. Such manual steps can make such approaches prohibitively expensive. Moreover, another limitation of such approaches is the fact that they can only be used to detect faults on the exception handling code after the system was implemented. Few development approaches have addressed this issue, such as [17], [22], but they are costly and heavily based on documentation, imposing an overhead that can be impeding to their use. Experiences [21] have shown that lightweight agile methodologies have been successfully used on the development of modern applications. Such methodologies are highly based on test automation and little documentation; in such agile development approaches, automated tests serve as live documentation. However, current agile methodologies do not explicitly take into account the testing of exceptional behavior.
In this paper, we propose an agile approach to specify and test the exceptional behavior of a system. In particular, we are interested in testing whether the paths traveled by exceptions are correct, from the points where they are thrown to their destinations. We have implemented this approach as an extension to the well-known JUnit framework. Tests for the exceptional behavior, called as exceptional tests from now on, are specified as regular JUnit tests with a small amount of extra information. An exceptional test can be used to specify possible exception paths and compare them with the paths that exceptions may travel at runtime. Our approach can be used to write tests: (i) before the system has been implemented using the concept of "test-first" development; or (ii) in latter development phases, to help in maintenance activities, in existing systems. We have applied the proposed approach to production systems written in Java and have uncovered twelve bugs, eight of them previously unknown.

The rest of the paper is organized as follows. Section II presents a motivating example. The example explains why JUnit, the most popular approach for the construction of automated tests cases, fails to detect bugs caused by exception propagation. Section V briefly describes related work. Section III presents our proposed approach. Section IV presents a preliminary evaluation and the last section rounds the paper.

II. A Motivating Example

In this section, we present, a simple illustrative, example to point out some limitations of pure JUnit Test Cases to detect exception handling bugs. Consider a layered information system composed by three layers: Data, Business, and GUI (Figure 1). An exception handling policy that should be obeyed in this application is the following: the exceptions signaled by Data Access Objects (defined on the Data layer) represent problems in attempts to access a database. They are subtypes of DAOException. These exceptions should be handled by Servlets defined on the GUI layer. In order to check whether this policy is obeyed, the developer could try to define a JUnit test. The following code snippet is a possible implementation for the test.

```java
public void testDAOEHPolicy (){
    Servlet s1 = new Servlet();
    s1.service();
}
```

If no instance of DAOException escapes from this test method, the tester may think that it was adequately handled by the Servlet. However, two faults may happen, which cannot be detected by this test case: (i) the instance of DAOException may be mistakenly caught by the Facade (an intermediate element between the signaler and the intended handler); (ii) the DAO may not signal the exception. In addition, a developer might want to test whether a method signals the exception when an erroneous condition is detected. JUnit supports the specification of test cases that expect an exception using the `expected` annotation:

```java
@Test(expected=DAOException.class)
public void testDAOEHPolicy (){
    Servlet s1 = new Servlet();
    s1.service();
}
```

In this case, the test will pass whenever it ends by signaling an exception of type DAOException. If an instance of DAOException is mistakenly captured by another catch block and no other exception is signaled, the test will fail, as expected. The same occurs if DAOException is not signaled. This approach is still limited, however, because it can hide subtle bugs. For example, if a component from the Data layer signals an exception representing a programming mistake, such as a null reference (NullPointerException), but that exception is captured by a generic handler, e.g., a catch clause for type Exception, which, in turn, signals DAOException, the test will still pass, but the programming mistake will be hidden. In fact, we have detected this kind of problem in existing applications (Section IV). Examination of the bug report systems of these applications reveals that they are considered bugs.

The JUnit framework does not provide a way of checking whether an intermediate element mistakenly handles the exception. This is understandable, since JUnit aims to support unit testing. Notwithstanding, when it comes to exceptions, unexpected global exception propagation can be a significant source of bugs [14], [16]. This is particularly noticeable for languages like Java and C#, where exceptions are objects that can be captured by type subsumption [16], that is, an exception handler for a type T captures exceptions of any type T′ that is a subtype of T. On the other hand, JUnit is the de facto standard for agile automated testing and has been extended, in some scenarios, to handle broader-scoped (e.g., integration) tests [4], [26]. Moreover, we believe that exception handling should be taken into account throughout all software development activities [17] and that test-driven development, in particular, and agile methods, in general, provide better support for this idea than a documentation-based development process. As a consequence, we consider that the JUnit approach to testing should be employed in other contexts, not only unit testing of normal behavior.

The bottom line is that there is a need of a testing approach to ease the development of exception handling test cases in the following ways: (i) the developer should be able of defining the complete or a partial exception path to be automatically checked during tests and deviations from this path should be considered bugs; (ii) the way exception tests are defined should be similar to the ones already defined by

1 http://junit.sourceforge.net/doc/faq/faq.htm#tests_8
developers using JUnit - to reduce the learning curve; and, finally, (iii) such tests should be used as live documentation concerning the exception flows of a program.

III. TESTING THE EXCEPTIONAL BEHAVIOR

We have devised an agile approach to test exceptional behavior of a system, which allows the global reasoning of the exception flows of an application, in other words, it allows the developer to specify and examine the paths traveled by exceptions from the points where they are thrown to their destinations.

A. Proposed Approach

The central idea is to establish the paths whereby the most relevant exceptions thrown must traverse. Exception paths are defined by means of semi-automated tests and runtime monitoring. Our approach supports the definition of tests cases to be built before the system is implemented or used as regression tests.

The exceptional behavior must be designed to conform with the test cases. Testers can specify what exceptions should be raised and how they flow among system components, in terms of the methods through which they pass and the order in which this occurs. This specification is part of the test implementation and, therefore, must be maintained by developers (otherwise, the tests will not pass), unlike documentation-based solutions, which can remain outdated. At runtime, the paths taken by exceptions are recorded and compared with the expected paths, specified in the tests.

Figure 2 illustrates the components of the proposed approach. In the figure, rectangles with rounded corners indicate activities, lozenge denotes alternatives, and remaining boxes represent artifacts. Continuous lines indicate ordering between activities and dashed lines indicate artifact dependences. First of all, the most important exception paths in the application must be selected (Section 3-B). These paths are the ones traversed by exceptions that represent application-specific situations where, due to external factors, the system is unable to provide its intended service. Since, in the early stages of development, remarkably little information about exception paths might be available, in our approach, it is possible to specify partial paths that can later be complemented with additional information. After that, test cases are written for all the paths devised in the first activity. We often write one test case per exception path, but it is possible to write a single test case to test a number of paths associated with the same exception. Test cases are then grouped in a test suite and executed against the system implementation. If all the tests pass, we consider that the exceptional behavior of the system adheres to its specification. Otherwise, inconsistencies were found. An inconsistency might stem from maintenance activities that changed exception propagation paths in a non-harmful way or from bugs. After the inconsistency is fixed, tests are run again and this cycle is repeated until the tests pass.

The specification of the expected path can use minimum or exact path semantics. Figure 3 illustrates the difference. The left-hand side of the figure depicts an actual exception path where an exception was raised by method d and handled in method a. In the case of minimum path semantics, all elements of the specified path must be part of the actual exception path, in the order in which they were specified. However, the actual path may contain more elements than the specified path. Figure 3-B presents examples of the specified minimum exception paths that match the presented actual path. Figure 3-C presents some invalid minimum paths. For exact path semantics, actual and specified path must be identical.

B. Selecting exceptional flows

Test cases are created to verify adherence to its specification at runtime for testing the exceptional behavior. Ideally
the test should be designed and executed for all exception flows available. This action is quite costly, so it is ideally important to select a reduced set of exception flows with high probability of bug occurrence.

Certain characteristics must be taken into account when specifying exceptional behavior by a test case. In the following, we list some potential candidate to an unexpected behavior of exception handling mechanism:

- Try-catch block that does not throw exception anymore.
- Handlers with generic exception type that remaps the caught exception into another exception type.
- Blocks that raise different exception types.
- Blocks that catch a large number of exception types.

C. Framework

We have extended the JUnit framework to support the proposed approach. This extension allows the definition of expected exception paths within a test case, in terms of raising, intermediate, and handling sites, and the exception that is raised for Java programs. Figure 4 presents a real test case example.

The expected path is an array of Strings (lines 13-18) that follows some conventions to indicate raising site (line 14), handling site (line 17), etc. The order of the elements in this array indicates the ordering of the exception path, starting from the exception and the raising site (always the first two items), all the way to the handling site. All the methods that appear in the following, e.g., exception(), and handlingSite(), are part of our extension to JUnit. The exception path must be set by invoking setExceptionPath() (line 21) before test case execution.

The testEHFlowDAOServlet() method (line 10) implements the actual test. It sets up any components that must be running before the test and executes the test. As in any functional testing approach, the test is responsible for triggering the exception condition that will result in the expected exception.

It is possible to redefine the behavior of the test result, for example, to prohibit, explicitly, an exception from traveling a certain path, by overriding the result() method. The following code snippet depicts one example of test case result redefinition:

```
// Super class method, optional override.
// Use to change the test result criteria.
public void result() {
    // Eg: If a specific exception occurs
    // test case should be fail
    assertTrue(!verifyResults());
}
```

The implemented extension allows the generation of a log file after tests execution. This file contains all the exception flow information produced by test case execution. This information is important because with it is possible to verify what has changed from one version to another one. The symbol # represents the exception type, the subsequent line represents where the exception was thrown, and after that, we have the intermediate sites. The symbol @ represents the catch site.

```
# pkg.MyException
pkg.raisingSite
pkg.intermediateSite1
pkg.intermediateSite2
@pkg.catchingSite
```

This extension also allows testing exceptions remap one exception type into another, see one example at Figure 5. This figure shows the execution of a(), b(), c() in this order. Method c() raises an IOException and propagates to method b(). Method b() catches this exception and remaps it into a MyException that is caught by method
Unfortunately, Junit does not support the definition of test cases to remap exception types.

D. Monitoring exceptions

We use Aspect Oriented Programming [10] to monitor exception propagation during test execution. The following code snippet captures the exception object. With this object and the local where the advice affected the code, it is possible to define the path traversed by the exception.

```
1- pointcut callexecutionGeneral():
2- handler(Throwable+);
3- before(Throwable e):
4- callexecutionGeneral() && args(e) {
5- //Before handling Throwable or
6- //subtypes, exception object is
7- //received without any code changes.
8- }
```

There are situations where an exception is thrown, but there is no handler available for handle this exception. For these cases another aspect is used:

```
1- pointcut callexecutionGeneralUnhandled():
2- call(* * . * (..));
3- after() throwing(Throwable e):
4- callexecutionGeneralUnhandled() {
5- //For each throwing exception,
6- //this aspect sends the method
7- //name and the exception to
8- //the junit extension.
9- }
```

These aspects are used to verify the adherence of the expected path to the actual one. If the actual and expected paths match, according to the specified semantics, the test passes.

E. Force exception

In some cases, it is difficult to reproduce the conditions that trigger the raising of an exception. This is often the case when an exception stems from external factors, such as I/O operations. It is possible to force an exception in a desired place at the exceptional path by using the @ForceException annotation in the test case definition (figure 4 lines 5 to 8). This annotation example enables the raise of java.io.IOException when the MyDAO.insertData(String) method is executed.

To force an exception we have developed a preprocessor that automatically generates exceptional aspect based on the test case annotation. This aspect forces the throwing of an exception. Developers only need to weave it [10] and do not need to have any familiarity with AOP. The following snippet code shows a sample aspect generated by the preprocessor.

```
1- public aspect MyTC {
2- pointcut tc():
3- execution(void MyDAO.insertData(String));
4- void around() throws
5- java.io.IOException :tc() {
6- throw new java.io.IOException();
7- }
8- }
```

This aspect replaces the implementation of MyDAO.insertData(String) method by the throwing of an IOException.

IV. Evaluation

We have evaluated the proposed approach by applying it in two real systems. Our idea was to simulate the creation and execution of tests to a system throughout its development. We looked for production quality Java open source applications with a large number of exception handlers and several versions available. Based on these criteria we have selected two open-source, production quality target systems with multiple versions and devised test cases for one of its initial, stable versions.

We then applied the same test cases to later versions of the same system. At all test cycle, if a test case failed, we checked if the fault actually existed or was a valid implementation changes caused, for example, by requirements modification, name changes, or elements removal. If such changes were verified, the test case was refactored. We believe this is one of the potential usage scenarios of the proposed approach. For this evaluation, we are interested only in bugs related to exception handling mechanism like wrong exception propagation, unhandled exceptions, etc. Other kinds of bugs are disregarded in this study.

The first target system was aTunes\(^2\), a powerful, full-featured, cross-platform player and manager, with audio cd rip frontend. This application has between 20K (initial version) and 50K (last evaluated version) lines of code, and has 31 available versions. In spite of its relatively small size, aTunes has a large user base (approx. 4.8K weekly downloads at sourceforge.net). For this system, we have created 17 test cases and found four bugs over six analyzed versions.

\(^2\)http://www.atunes.org/
The second target system was jEdit\(^3\) system. jEdit is a programmer’s text editor written in Java. It uses the Swing toolkit for the GUI and can be configured as a rather powerful IDE through the use of its plug-in architecture. Our intention with this system was to evaluate the proposed approach applied to a larger and more complex system. The last jEdit version is bigger than 100k lines of code. Altogether 15 test cases were created, eight bugs found, and five versions analyzed.

For both systems, some intermediate versions were not tested due to minor changes in the exception handling code. Table I summarizes test results for aTunes and table II for jEdit systems. This table shows tests that passed without restrictions (PASSED), with changing of intermediate sites (PASSED-CALL), with changing of method or package names (PASSED-PKG). In the latter two, test cases must be updated in order to reflect the correct exceptional flows.

### A. aTunes evaluation

The first step for this evaluation was the choice of which version should be chosen as base version. We have started with version 1.5 based on the stability of the code base (there were major changes between 1.3 and 1.4). We then proceeded to specify the expected exception paths. We have created one test case for each exception path that has at least two nodes that are internal to the application (none is in an external library). The test cases made initially for version 1.5 served as a basis for the other versions. Because of code changes between versions, some test cases had to be redefined and new ones were created. These changes were triggered by changes in the implementation of exceptional behavior or changes in package and method names in some

\(^3\)http://www.jedit.org/
cases, public methods and some classes were removed. In these cases, we either modified or removed test cases.

We tested versions 1.5, 1.6, 1.9, 1.10, 1.12, and 1.13 of aTunes.

In version 1.9, we found the first bug. Test case TC11 that should raise java.io.FileNotFoundException instead. After manual inspection, we found that AudioFilePictureUtils.savePictureToFile() method always returns a null object. This issue was fixed in later versions of aTunes system.

In version 1.10, found a potential source of bugs[16] for future versions of the system. By inspecting the aTunes bug database, we discovered that neither of these problems had been previously reported. In addition, we found two bugs in subsequent versions that had been previously reported. Since version 1.9 until 1.13, test case TC17 revealed unhandled exception. In the execution flow started by test case startup, one method returns a null object. This object is accessed generating NullPointerException. Since the exception is not handled, forces the closing of its execution. Code above shows the issue. Method savePictureToFile() (line 1) call getInsidePicture() (line 10) that return null (line 15). The null object is received by image (line 5) after that image.getImage() is called (line 7) causing the exception.

```java
private static void savePictureToFile()
 throws ... {
    ImageIcon image = getInsidePicture(song);
    PictureExporter.savePicture(image.getImage(),
 file.getAbsolutePath());
}
```

```java
private static ImageIcon getInsidePicture
 (AudioFile file) {
    FileInputStream stream = null;
    try {
 return null;
    } catch (Exception e) {
 finally {
 ClosingUtils.close(stream);
 }
```

**B. jEdit evaluation**

We decided to evaluate a second system, larger and with more code related to exception handling. Initially we specify tests for jEdit 4.0 final version. Altogether 15 test cases were created, and eight bugs were found over five evaluated versions (4.0, 4.1, 4.2, 4.3 and 4.3.1). Two test cases were created based on bugs previously reported. After test case specification, all test cases were ran. The first bug was found in test case TC02. This bug was caused by throwing of NullPointerException exception. This bug had been reported previously at jEdit bug database. After the execution of TC02, we update the test specification so, if, in the next version, the same bug occurs, this test should fail. In all other versions evaluated, we observe, at TC02, the occurrence of the same issue. The second bug, at TC04, is due to the occurrence of ArrayIndexOutOfBoundsException. Similarly to the first, it was previously known. This bug was observed up to jEdit 4.1, after this, the target test code was removed. The third bug was found in TC07. EvalError exception was expected, but ClassCastException was thrown. The fourth bug was found in jEdit 4.2. At TC05 the design path was:

```java
1- String [] trace = new String []{
2- exception("bsh.EvalError"),
3- raiseSite("bsh.Name.toClass"),
4- intermediateSite("bsh.BSHAmbiguousName.toClass"),
5- intermediateSite("bsh.BSHTypedVariableDeclaration.eval"),
6- intermediateSite("bsh.BSHTypedVariableDeclaration.eval"),
7- intermediateSite("bsh.Interpreter.eval"),
8- intermediateSite("bsh.Interpreter.source"),
9- catchSite ("bsh.Interpreter.main"));
10-}
```

But in test case executing bsh.TargetError exception was thrown. The following log shows the executed exception path:

```java
#Bsh.TargetError
bsh.Interpreter.eval
bsh.Interpreter.source
@bsh.Interpreter.main
```

The fifth bug was found at TC10 version 4.2. A ClassCastException was thrown. The next bug (TC11) occurs because an EvalError exception was expected (line 2), but TargetError (6 lines) was thrown on test case execution.

```java
1- String [] trace = new String []{
2- exception(bsh.EvalError"),
3- raiseSite (bsh.EvalError"),
4- catchSite (test.TC11.startup ")
5- );
```

**Log of TC11 version 4.2:**

```java
6- #bsh.TargetError
7- bsh.Interpreter.eval
8- test.TC11.startup
```

The last two bugs were found in TC14 and TC15. The running of both test cases ending with the raising of ClassCastException. The approach proved effective for finding bugs related to the usage of exception handling.

---

4http://sourceforge.net/tracker/?atid=821812&group_id=161929

5http://sourceforge.net/tracker/?group_id=588&atid=100588
mechanisms. Altogether 32 test cases were created, and 37.5% of it have been failed. It is also effective to indicate code changes. 12.4% of valid tests performed indicate code changes related to exceptional flows modifications like method/package names or by the addition/subtraction of intermediate sites. These changes do not mean that test case have failed once they help developers to understand code evolution. In this case, forcing the update of the test case guarantees the documentation is always up to date. The definition is straightforward, with two steps (definition of exceptional flows and startup) its possible to create a test case.

V. RELATED WORK

In this section, we present a set of research work directly related to our own. They are organized in three categories: (i) static verification approaches for exception handling code base; (ii) testing approaches for exception handling code base; (iii) development processes that integrate exceptions into the entire software development life cycle.

A. Static Verification Approaches

In the literature, several papers propose solutions based on static analysis that allow exception flow analysis[6], [16], [5], [8], [14]. An exception flow analysis tool traverses the program call graph and estimates the paths that exceptions will travel at runtime. Chang et al. [5] use static analysis to estimate the exception flows to detect unnecessary or general exception specifications (throws clauses) or handlers in Java programs. Fühndrich et al. [6] have developed a toolkit to discover uncaught exceptions in ML programs using the constraint-based program analysis. Robillard and Murphy proposed a design approach for simplifying the exception structure in Java programs [16] and developed the JEX tool. This tool extracts information about the flow of exceptions, providing a view of the actual exception types that might arise at different program points and of the handlers that are present. Fu and Ryder [8] proposed the idea of exception chain analysis as a means to narrow down the results produced by tools such as JEX. Exception chain analysis attempts to reduce the number of reported exception propagation paths by merging exception paths that are related. Moreover, few more other tools were created in order to facilitate understanding of exceptional flow analysis. JEXVIS [19] uses the textual information to generate an interactive visualization of exception structure. EXTEST [8] presents a tree view of exception handlers, their triggers, and their propagation paths. EXPO [20] computes exception handling statistics and visualizes the context of throw-catch pairs as a flow graph. The ENHANCE tool [11] presents four different views of exceptions in a system. The main limitation of static analysis approaches, as well as the associated visualization tools, is that they support discovery but not enforcement of exception paths. If developers employ such a tool to understand the exception structure of a system and later the system is modified, a large amount of effort might be spent discovering how these changes affected exception paths. At the same time, due to factors such as inheritance and polymorphism, these tools often report many false positives. Finally, static analysis is only useful when the system has already been implemented. Therefore, they cannot help in the specification and design activities.

B. Testing Approaches

JUnit framework⁶, a widely used testing framework, allows developers to check whether a piece of code throws a given exception during the execution of a test case. However, it does not offer any construct to enable a developer checking either a specific exception path occurred during the execution of a test case, or whether a specific handler catches a given exception. Sales et al. [18], [27] propose an approach to support the definition and test of what they call exception handling contracts. An exception handling contract specifies the elements responsible for signaling exceptions and the elements responsible for handling them. In their approach JUnit tests are partially generated from such contracts. Our proposed approach differs from this previous work because we can specify both the location where the program throws an exception and where it will be treated. This approach takes into account the fact that the overall effect of an exceptional occurrence should consider the intermediate levels on which the exception may propagate before reaching its handler. Hence, our approach enables the specification on which intermediate elements should be present on an exception path. Moreover, we added new constructs to JUnit framework in order to allow the definition and checking of the exception path on a test method body.

C. Development Processes

Few approaches have been proposed aiming at the definition of the system exception handling behavior throughout all the software development activities prior to implementation. Some of them incorporate exception handling-related activities into existing software development methodologies [17], while others have targeted specific development activities [2]. All of them, however, have a strong emphasis in producing documentation about the exceptional behavior. In our approach, we consider such characteristic to be their greatest limitation. It is well-known that developers often do not keep documentation up-to-date with the system implementation [15]. For that reason, our approach relies on the definition of automatic tests in order to specify the behavior of the exception handling code, which will serve as a live documentation.

⁶http://www.junit.org/
VI. CONCLUSIONS AND FUTURE WORK

Software engineers are faced with the challenging task of developing, debugging, and maintaining software systems that contain exception handling constructs. Dealing with exceptions since the start of software development allows for the software engineers to produce more robust exception handling code. In some cases, we are faced with existing systems that require maintenance. Exceptions complicate programs particularly because they require a global view of the system – to understand how exceptions affect program execution; it is not enough to look at specific points in the program’s source code. It is important to determine which exceptions flow to a point in a program and where they come from. Unfortunately, existing exception handling mechanisms have a local focus.

For this, we have proposed a new agile approach to test exceptional behavior. To support this approach, we developed an extension of the JUnit framework. We have shown the proposed testing approach helped us to uncover a number of bugs to two medium-sized production systems.

We foresee three lines of future work. First, we intend to conduct an additional case study, to gather more evidence about the benefits that our approach brings to maintenance activities. Second, we would like to conduct a controlled experiment with two groups of developers building a system from scratch, one of them using our testing approach and the other one not using it. Finally, we would like to evaluate how exceptional behavior test cases can complement existing static analysis techniques.

VII. ACKNOWLEDGMENTS

We would like to thank the anonymous referees, who helped to improve this paper. Fernando is supported by CNPq (308383/2008-7 and 475157/2010-9) and FACEPE (APQ-0395-1.03/10), Sérgio is supported by CNPq (305085/2010-7) and FACEPE (APQ-0093-1.03/08), Nelio is supported by FAPERN (PPP-III/79-2009). This work is partially supported by INES (CNPq 573964/2008-4 and FACEPE APQ-1037-1.03/08).

REFERENCES


7The jUnitE extension and tests used in this evaluation is available at: https://sites.google.com/a/cin.ufpe.br/castor/jUnitE


