INCREASING UNIT TEST RESILIENCE BY DECREASING POINTCUT FRAGILITY

Michal Hucko,
Valentino Vranić*

Institute of Informatics, Information Systems and Software Engineering,
Faculty of Informatics and Information Technologies,
Slovak University of Technology in Bratislava, Bratislava, Slovakia

Abstract:
By operating at a very detailed level, unit tests are very susceptible to changes in production code. Writing unit tests in aspect-oriented programming can help with their maintainability. However, the existing approaches do not take into account so-called pointcut fragility: a failure to address the intended join points due to small changes in the base code. An approach to increasing unit test resilience to changes in production code by decreasing pointcut fragility is proposed in this paper. The approach is implemented in AspectJ with JUnit used as a test oracle. The approach has been evaluated on several scenarios encompassing typical code modification that render unusable the tests written in a simple object-oriented way. The approach proposed in this paper managed to make the test resilient to the most of the changes introduced by these scenarios.

Keywords:
software testing, aspect-oriented programming, pointcut fragility, AspectJ, Java.

1. INTRODUCTION

Writing production code requires writing huge amounts of testing code. In particular, this is true for so-called unit tests: the tests that address the smallest testable parts. Test driven development practically requires to cover the whole production code with unit tests.

By operating at a very detailed level, unit tests are very susceptible to changes in production code. Even minor changes in production code often make unit tests obsolete or even pointless.

Writing unit tests in aspect-oriented programming can help with their maintainability. Aspect-oriented programming paradigm with its separation of crosscutting concerns suits well this purpose. The idea is that the aspects can be used to throw runtime exceptions which can be handled by the test oracles such as JUnit [1]–[3]. With aspect-oriented programming, the tests are maintained completely outside of the production code, being attached to it at the points they need to introspect, known as *join points*. These are specified declaratively as sets of well-defined points in program execution by constructs called *pointcuts*. However, the existing approaches do not take into account so-called *pointcut fragility*: a failure to address the intended join points due to small changes in the base code.

Correspondence:
Valentino Vranić

e-mail:
vranic@stuba.sk

DOI: 10.15308/Sinteza-2019-369-373

In this paper, we will look at the possibilities of increasing unit test resilience to changes in production code by decreasing pointcut fragility. For this, we will use the AspectJ programming language, a widely known embodiment of aspect-oriented programming based on Java. The rest of the paper is structured as follows. Section 2 explains pointcut fragility. Section 3 proposes an approach to increasing unit test resilience to changes in production code by decreasing pointcut fragility. Section 4 presents the implementation, and Section 5 resents the evaluation of the approach. Section 6 discusses related work. Section 7 concludes the paper.

2. POINTCUT FRAGILITY AND TESTING

Consider this simple aspect written in AspectJ:

```
public aspect SampleAspect {
   around(Message msg):
   call(init Counter.countLetters(..)) && args(msg) {
      // Process message
   }
}
```

Even small changes to the code this aspect affects, such as changing the return value type of the `countLetters()` method, can make its pointcut fail to address the corresponding join points. This pointcut breaks on a small change: it's fragile.

Koppen and Störzer identified the following situations that cause fragile pointcuts in AspectJ to break [4]:

1. Renaming classes, fields, and methods
2. Moving a method or class, which invalidates the pointcuts based often used lexical primitive pointcuts `within()` and `withincode()`
3. Adding or removing classes, fields, and methods, which results either in making pointcuts fail to cover new elements or in making them target what does not exist any more

Writing tests capable of dealing with dynamically evolving systems is a challenge [5].

3. APPROACH

Here, the actual approach to increasing unit test resilience to changes in production code by decreasing pointcut fragility is proposed. The approach assumes the unit tests are written in AspectJ, while the production code is written in Java.

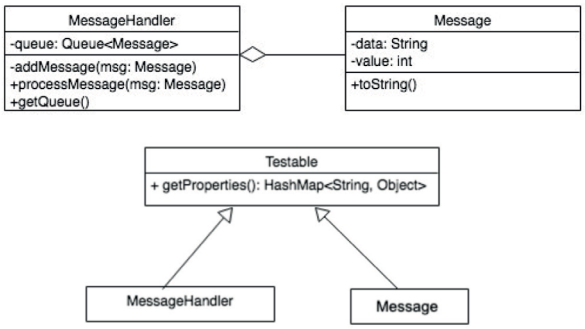

Fig. 1. The classes being tested inherit from Testable

First of all, if we are able to identify all the classes and methods that need to be tested and if we can provide them with good and stable structure and names as we design them, we can overcome future problems with fragile pointcuts. Of course, we can hardly predict all the changes to come.

In most cases, we are simply faced with the code to be tested without the possibility to redesign it to ease the testing. Even under such circumstances, we can identify the classes to be tested. We need a mechanism to put all these classes under a common handle. One way of doing this is to make the tested classes inherit from a special common class. This is exactly what is being used in the approach proposed in this paper. This common supertype for all classes to be tested is named Testable. The easiest way to ensure inheritance without having to modify the production code is to use the declare parents AspectJ inter-type declaration. Thanks to inheritance, the testing aspect can refer to all these classes regardless of their names and how they change over time. For Testable, an interface could have been used instead of a class, which would spare the only possible extends relationship at the classes to be tested. Pointcuts are established around the Testable class as a common supertype with a reasonable use of wildcards in signatures within the pointcuts. Java reflection is used to access necessary objects within advice bodies. All this accommodates future changes in method signatures.

The actual tests are implemented as aspects. As with all tests, the tests implemented as aspects signal undesired situations. The signaling is implemented so that the aspect that implements a test raises a dedicated exception denoted as `TestingException`. This exception is derived from the original Java Exception class and it points to the place where its instance occurred.

The test oracle is running the code which is being tested. After the target pointcut has been reached, the testing aspect-oriented code is executed. If this code does not raise a `TestingException`, the test has passed.

4. IMPLEMENTATION

Consider the situation depicted in the upper part of Figure 1. The `MessageHandler` class represents a message handling unit which handles incoming messages through its `processMessage()` method.

As is depicted in the lower part of Figure 1, both of these classes should inherit from the Testable class, which implements the class reflection through the `getProperties()` method. This method returns a `HashMap` of properties for a given instance.

As a test oracle, the implementation presented here uses the JUnit framework. Each test calls some method of one of the classes being tested and catches a `TestingException`. If the exception occurs, the test is considered to have failed.

Consider these two unit tests:

1. Before adding a `Messageobject` to the `MessageHandler` queue, the actual queue object must be initialized
2. The messages with a specific value assigned are not to be added to the queue within the `processMessage()` method call

Their implementation could look like this:

```
public aspect MessageHandlerTestingAspect {

   protected boolean init = false;

   declare parents: MessageHandler || Message
         extends Testable;

   pointcut initReached(Testable mh): target(mh)
     && call(* *Handler.*init*(..));

pointcut addReachedHandler(Testable mh):
  target(mh) && call(* *Handler.*add*(..));

pointcut addReachedMessage(Testable msg):
   call(* *Handler.*add*(..)) && args(msg);

after(Testable mh): initReached(mh) {
   init = true;
}

   before(Testable mh) throws TestingException:
      addReachedHandler(mh) {
      if (!init) {
         TestingException ex =
            new TestingException("Not Initialized");

            ex.setSource("Queue was not initialized.");

            throw ex;
      }
   }

   void around(Testable msg) throws TestingException:
      addReachedMessage(msg) {
      Object value = null;

      try {
         value = msg.getProperties().get("value");
      } catch (IllegalArgumentException |
         IllegalAccessException e) {
      }

      if ((int) value != 2) {
         proceed(msg);
      } else {
         TestingException ex =
            new TestingException("Not Initialized");

      ex.setSource(
         "Trying to add a forbidden message.");

      throw ex;
      }
   }
}
```

The first test is implemented by the `initReached()` and `addReachedHandler()` pointcuts and first two pieces of advice (of the after and before type). The second test is implemented by the `addReachedMessage()` pointcut and the remaining piece of around advice. The declare parents statement is used to introduce the inheritance.

5. EVALUATION

The increased unit test resilience to changes has been evaluated on several scenarios applied to the situation presented in the previous section:

1. Changing the `MessageHandler` class name. In this scenario we changed the `MessageHandler` class name to `Handler` and to `SomeThing`. The testing aspects were able to handle the names derived from `Handler`. In case of `SomeThing`, they failed.
2. Changing the name of the `Message` class. In this scenario we changed the name of `Event`. Because our aspects are using wildcards, all the modifications to parameter names were fine.
3. Adding an extra argument to the `processMessage()` method. We added an integer argument named `j` to the method call. The testing aspects handled this situation.
4. Renaming an argument in the `processMessage()`. We renamed the `msg` argument to `msg2`. The testing aspects handled this situation, too.
5. Renaming the `init()` method. We renamed the `init()` method to `initQueue()` and `prepareQueue()`. The same problem occurred as in the first scenario. Since we rely on the method name to contain the `init` string, we are unable to handle situations where this string is not present.

Each of these scenarios would make the tests written in a simple object-oriented way obsolete or broken. Here, it has been demonstrated that just by adding wildcards to pointcut definitions and making all tested classes inherit from the Testable class using the corresponding AspectJ inter-type declaration, the testing aspects can be made more resilient to code refactoring changes.

6. RELATED WORK

Xu and Yang proposed a method for unit testing using aspects [2]. They identified that the separation of crosscutting concerns suites well to unit testing. They used so-called application specific aspects for testing functionality of a program. They also used testing aspects to raise runtime exceptions. They implemented the tests in the Aspect-Oriented Test Description Language (AOTDL). AOTDL code can be translated by JAOUT/translator to AspectJ code. In the end, they used JAOUT/translator for automatic generation of JUnit test classes. These play a role of test oracles that handle test exceptions from the testing aspects.

Xu and Yang also presented JAOUT as a tool for automatic generation of unit tests using testing aspects [3]. Combining both of their approaches, their tool was able to test the code with given testing aspects written in AOTDL.

Sakurai and Masuhara proposed test based pointcuts [1]. They used unit tests to specify join points at which the actual aspects are being weaved. The whole process consists of two main steps. First, unit tests (implemented using JUnit) are executed and the sequences of the join points they address are being recorded. Afterwards, when one of the recorded sequence matches, the corresponding aspect is weaved. This approach mitigates the effect of pointcut fragility. Sakurai and Masuhara used a special notation like:

```
test(get(* Fixtures.invalidUser));
```

for specification of pointcuts. They used the AspectBench Compiler to develop their prototype.

Using aspect-oriented programming for testing is widely present in the JBoss server. The principle is the same as with the previous approaches. The aspects used for testing the functionality are throwing exceptions in case of failure and the test oracles are catching them. According to the documentation,[1] aspects are defined in separate XML files and they can be added or replaced at runtime. Mock objects are used for actual testing, and the JUnit framework is used as a test oracle.

Hughes et al. [6] reported using aspect-oriented programming to test a distributed system called the AGnuS. In their work, they identified several key problems in dealing with software testing. One of them is the reuse of testing code in other applications, which is very close to the objective of the approach proposed in this paper. For this, Hughes et al. enriched AspectJ syntax with special tags which use Java reflection API. The approach proposed in this paper does not require any changes to the underlying programming languages, i.e., Java and AspectJ.

7. CONCLUSION AND FURTHER WORK

An approach to increasing unit test resilience to changes in production code by decreasing pointcut fragility has been proposed in this paper. Pointcut fragility is decreased by imposing a common supertype on tested classes and, consequently, by establishing pointcuts around this common supertype with a reasonable use of wildcards in signatures within the pointcuts accompanied by using reflection to access necessary objects within advice bodies to accommodate future changes in method signatures. The approach is implemented in AspectJ with JUnit used as a test oracle.

The approach has been evaluated on several scenarios encompassing typical code modification that render unusable the tests written in a common object-oriented way. The approach proposed in this paper managed to make the test resilient to the most of the changes that have been made to the production code.

The approach could be extended to employ a synonym dictionary to generate additional (predicted) possibilities and build them into pointcut declarations. JAOUT tool [3] could be used to automate this process. However, this needs to be balanced, as extensive pointcuts may obscure the intent. Furthermore, automatically recorded tester actions over the system being tested [7] could be used to interactively generate pointcuts.

3D visualization of software models [8]–[10], along with virtual reality [11], [12] could be used to model and generate more robust pointcuts. More robust pointcuts would be of help in aspect-oriented refactoring [13], in capturing events in complex event processing [14], [15], and in defining language semantics through aspects [16].

1 http://docs.jboss.org/aop/1.3/aspect-framework/userguide/en/html/

ACKNOWLEDGMENT

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/0759/19, by the Slovak Research and Development Agency under the contract No. APVV-15-0508, by the education and research development project "STU as a digital leader," project no. 002STU-2-1/2018 by the Ministry of Education, Science, Research and Sport of the Slovak Republic, and by the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

REFERENCES

[1] K. Sakurai and H. Masuhara, "Test-based pointcuts: a robust pointcut mechanism based on unit test cases for software evolution," in Proceedings of 3rd Workshop on Linking Aspect Technology and Evolution, Vancouver, British Columbia, Canada, ACM, 2007.

[2] G. Xu and Z. Yang, "A novel approach to unit testing: the aspect-oriented way," in Proceedings of 2004 International Symposium on Future Software Technology, ISICT '04, Las Vegas, Nevada, USA, 2004.

[3] G. Xu, Z. Yang, H. Huang, Q. Chen, L. Chen, and F. Xu, "JAOUT: automated generation of aspect-oriented unit test." in 11th Asia-Pacific Software Engineering Conference, APSEC 2004, Busan, Korea, IEEE, 2004.

[4] C. Koppen and M. Störzer, "PCDiff: attacking the fragile pointcut problem," in 1st European Interactive Workshop on Aspects in Software, EIWAS 2004, Berlin, Germany, 2004.

[5] A. Bertolino, "Software testing research: achievements, challenges, dreams." in Proceedings of 2007 Workshop on the Future of Software Engineering, FOSE '07, part of ICSE 2007, Minneapolis, MN, USA, IEEE CS, 2007.

[6] D. Hughes, P. Greenwood, and L. Blair, "Aspect testing framework." in Proceedings of FMOODS/DAIS 2003 Student Workshop, Paris, France, 2003.

[7] K. Frajták, M. Bureš, and I. Jelínek, "Exploratory testing supported by automated reengineering of model of the system under test," Cluster Computing, vol. 20, no. 1, pp. 855–865, 2017.

[8] M. Ferenc, I. Polášek, and J. Vincúr, "Collaborative modeling and visualisation of software Systems using multidimensional UML," in Proceedings of 5th IEEE Working Conference on Software Visualization, VISSOFT 2017, Shangai, China, IEEE , 2017.

[9] L. Gregorovič and I. Polášek, "Analysis and design of object-oriented software using multidimensional UML," in Proceedings of 15th International Conference on Knowledge Technologies and Data-Driven Business, Graz, Austria, ACM, 2015.

[10] L. Gregorovič, I. Polášek, and Branislav Sobota, "Software model creation with multidimensional UML," in Proceedings of 9th IFIP WG 8.9 Working Conference, CONFENIS 2015, part of WCC 2015, Daejeon, Korea, LNCS 9357, Springer, 2015.

[11] J. Vincúr, P. Návrat, and I. Polášek, "VR City: software analysis in virtual reality environment," in IEEE International Conference on Software Quality, Reliability and Security, QRS 2017, Prague, Czech Republic, IEEE, 2017.

[12] J. Vincúr, I. Polášek, and P. Návrat, "Searching and exploring software repositories in virtual reality," in Proceedings of ACM Symposium on Virtual Reality Software and Technology, VRST 2017, Gothenburg, Sweden, ACM, 2017.

[13] R. Pipík and I. Polášek, "Semi-automatic refactoring to aspect-oriented platform," in Proceedings of 14th IEEE International Symposium on Computational Intelligence and Informatics, Budapest, IEEE, 2013.

[14] J. Lang, M. Jantošovič, and I. Polášek, "Re-usability in complex event pattern monitoring," in Proceedings of IEEE 10th Jubilee International Symposium on Aplied Machine Intelligence and Informatics, Herľany, Slovakia, pp. 265–270, IEEE, 2012.

[15] J. Lang and J. Janík, "Reactive distributed system modeling supported by complex event processing," in Proceedings of ECBS-EERC 2013, 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems, Budapest, Hungary, IEEE CS, 2013.

[16] J. Porubän, M. Sabo, J. Kollár, and M. Mernik, "Abstract syntax driven language development: defining language semantics through aspects," in Proceedings of International Workshop on Formalization of Modeling Languages, FML '10, ECOOP 2010, Maribor, Slovenia, ACM, 2010.