STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Special Issue 1, 2014
10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, May 21-25, 2014

THE EFFECTS OF USING EXCEPTION HANDLING ON SOFTWARE COMPLEXITY

GERGELY NAGY AND ZOLTÁN PORKOLÁB

Abstract. Exception handling is the definitive way to handle errors of any kind and exceptional circumstances in modern software. There has been a long way before software methodlogy arrived to creating and using the notion of *exceptions*. We automatically assume that using exception handling makes our software more readable, more maintainable and easier to understand – i.e. less complex than when we use any other error management (let it be using return values, ERRNO or any other kind). Is this really the case?

Measuring *software complexity* can be done using software metrics. There are several trivial, well-known candidates – lines of code, cyclomatic complexity or McCabe-metrics and A-V – for this purpose, however these metrics do not measure exception contructs, therefor their usage can lead to distorted results[12]. In this paper, we extend the definitions of two metrics to the case of exceptions and analyze how these extensions affects these metrics on different error handling constructs. The extension are validated by the conformance to Weyuker's axioms and by real-world examples. We also examine industrial-sized software to prove that our definitions have no negative effect on the complexity measured by these metrics.

1. Introduction

Exceptions are the definitive way to handle errors or other exceptional circumstances in modern software. The goal of using these techniques is to make software more readable, more easily apprehensible and easier to maintain while supporting a wide range of functionality. The notion of exception handling was introduced in the 1960s in PL/I and was made more popular and usable in CLU[2]. Since the dawn of object oriented programming, exception handling has been used widely in production software.

Received by the editors: May 1, 2014.

2010 *Mathematics Subject Classification.* 68N19.

1998 *CR Categories and Descriptors.* D.1.5 [**Software**]: Programming techniques – *Object-oriented programming*.

Key words and phrases. Exceptions, software metrics.

Software metrics are used to measure a given property of a piece of software with an absoulte numeric value. Applying metrics to software can lead to better expression of the properties of the software, they can make planning lifecycle, time and resources required for creating and maintaining the software product easier. Based on this, the properties they measure can range from simply length to how complex the task to test or understand software is.

The shortcoming of these metrics is that they simply skip exception handling pieces of code from their analysis, thus we have no solid understanding of how it affects complexity. Our intuition suggests however that they have a considerable effect. There has been brought up an interesting problem in the Guru of The Week[5]: one needed to find all the possible execution branches in 4 lines of C++ code. According to their classifications, people who find 3 at most are average programmers, who find between 4 and 14 are exception-aware and the ones finding all 23 are exceptionally knowledgable. The steep increase in the number of execution branches suggests our intuition regarding exception handling is correct: the common aggreement is that execution branches highly correlate to software complexity.

In this paper we analyse how using modern error handling techniques – exception handling – affects software complexity. During our analysis we make use of complexity measures that are the conclusive units for this type of investigation, although they are not usually defined for exception handling. Our results are included for the A-V[4] and cyclomatic complexity[3] measures.

This paper is organised as follows: in section 2 we give a short introduction to the A-V metrics, then, in section 3 we extend cyclomatic complexity and the A-V metrics to the case of exception handling. In section 4 we examine how Weyuker's axioms for complexity measures hold with the extended definitions. We evaluate our extensions with synthetic examples as well as analysing a real-world application, Apache Tomcat[14] version 6 in section 5. Our paper concludes in section 6.

2. Introducing The A-V Metrics

The basis of our analysis is the A-V software complexity measure, that has been defined in 2002[4]. The need for this new metrics originates in the appearance and wide adoption of object-oriented and multiparadigm methodologies. While most complexity metrics can be used to analyse structured programs, in the last decade they have been proven to be unsatisfactory. Cyclomatic complexity, for example, was defined in 1976 with Fortran programs in focus[3]. Software development and languages have changed so much rendering these metrics unmeaningful for most usage nowadays. In this section

we will give a short introduction to A-V and will define terms and methods required for our extension.

The A-V builds on McCabe's basic thoughts. The basis for its calculation is the same control flow graph[8], but instead of using cyclomatic numbers to describe this graph, it defines nested deepness for predicates. Later researches, however showed the importance of nesting deepness of control statements[9][10].

Definition 2.1 (Nested deepness, ND)**.** This number represents how many predicates are in scope for a given statement in the program. The formal definition of nested deepness can be found in [13].

Using nested deepness directly describes how many predicates and branches need to be understood for a statement in the code: the more branches and loops you need to apprehend, the more predicates and variables you need to keep track of.

Using data is the central matter for almost all software. This trend has been invigorating with the spread of the object-oriented paradigm, where the core thoughts revolve around objects, how they are represented and what operations one can apply to them. These factors led to the necessity of considering how data is handled when one argues about software complexity. The A-V adheres to this trend with extending the control flow graph with data access nodes. Edges to these nodes are present from control flow nodes, when a variable is either written or read. It also allows counting these multiple times between the same nodes. The weight of these data access nodes is always given by the nested deepness of the statement they are connected to.

These consideration lead us to the following definitions:

Definition 2.2 (A-V measure of a method f)**.**

$$\sum_{s \in \{\text{Statements of } f\}} ND(s) + \sum_{v \in \{\text{Variables of } f\}} ND(v)$$

Definition 2.3 (A-V measure of a class C)**.**

$$\sum_{f \in \{\text{Method of } C\}} AV(f) + |\{v : V \text{ is a member in } C\}|$$

The above definitions make it clear that the A-V metrics is a natural response to the issues of cyclomatic complexity. Measuring complexity using these methods leads to a better description of modern software and also provides a solid base for our own analysis.

3. Extending Metrics for Exception Handling

In this section we propose extensions to the cyclomatic complexity and A-V metrics for the case of exception handling. We start with cyclomatic complexity, because it is the basis of the A-V metrics, then we extended the latter measure as well.

The basis for the metrics defined by McCabe is the role of predicates in the code, because these predicates set the execution path of the given code block. We need to introduce the notion of catching and throwing exceptions to this model. A trivial consideration is to think of all `catch` blocks after a `throw` as predicates: if there has been an exception, these `catch` blocks are predicates on the type of the exception, hence in the extension each `catch` branch increases complexity by 1. Throwing exceptions is not this trivial. All the conditional statements that lead to a `throw` are of course accounted for, however we disregard them when calling these functions. The reason for this is that these function calls do not increase the number of predicates in client code, thus they should not be considered based on the original thoughts of McCabe. If these functions are part of our code base, they will increase the complexity of our software.

The original A-V metrics uses the idea of nested deepness to better characterize complexity, furthermore it registers data access points in the control flow graph, because data is becoming the central concept of our software. The extension we propose takes steps further along these lines; it tries to be faithful to the prime considerations of the A-V measure. We utilize the notion of nested deepness as well as extend the data graph with exceptions.

Most programming languages represent exceptions with a typed object. We can safely ignore the fact in our analysis if these types have to extend one root exception type (e.g. most JVM-based languages), thus we make no differences to the original A-V when it comes to creating, throwing or catching exception objects. It will be noted as a simple data access, considering the nested deepness of the statement. The call to the constructor might involve other variables $1..n$, but these would be considered by the original A-V as well.

The most fundamental concept of A-V is to consider all decisions that need to be made leading to a given point in the program. This is handled by counting the number of predicates to which a statement belongs. Using this concept we can better describe throwing exceptions: we consider these statements with their nested deepness; apart from explicit `throw` statements, we also consider function calls as well, because they can potentially throw exceptions. If a function can throw more than one exception, we register all of them using the nested deepness of the call. The reason for this is that one needs to understand all of the conditions that can result in throwing exceptions and

that increases the number of test cases and complexity in general. Of course, `throw` statements should be counted only once, since they can only create one type of exception – they are only responsible of describing one type of exceptional condition.

Catching exceptions in client code happens in the same method in which they were raised often times, although there can be significantly complex code between these. It can easily happen that there are several branches that need to be evaluated, and this aggravates complexity. Because of these arguments, we consider the difference of nested deepness between catching and throwing an exception. Amidst a pair of a `try-catch` there can be multiple levels of checks that need to be understood. We naturally adhere to the semantics of programming languages by matching explicit and implicit `throw`s with their `catch`es and counting these. These pairs can always made unambigously – much like balanced parantheses.

Definition 3.1 (A-V measure of the exception e)**.**

$$\Delta ND(e) = ND(throw(e)) - ND(catch(e))$$

Contrary to client code, in libraries we cannot properly deal with an exception at the point of creation, making throwing and catching an exception far apart. To express this in our extension, we count all non-caught exceptions in the given method weighed by their nested deepness, and add it to the complexity of the method. The sole reason for this is that developers of the library need to understand the complexity that leads to raising an exception, but this increases complexity at the point of use of the function.

With these deliberations we arrive to the following definitions:

Definition 3.2 (Extended A-V measure of the function f)**.**

$$AV(f) + \sum_{e \in \{\text{Thrown exceptions of } f\}} ND(e) + \sum_{e \in \{\text{Exceptions of } f\}} \Delta ND(e) + \\ + \sum_{e \in \{\text{Uncaught exceptions of } f\}} ND(e)$$

Applying these definitions and methods we try to give an answer on how using exceptions should be considered in software metrics, so we can better account for software complexity.

4. Weyuker's Axioms for Software Complexity Measures

The original purpose of Weyuker's axioms[6] is to filter out all trivially wrong metrics that will be proven to be usuless in any real application. These statements are simple sanity checks against newly defined metrics. There have

been examples created though to show that there can be trivial metrics that meet all requirements[1] without any meaningfulness, although they still stand as the standard for software metrics. To show that we meet all of the axioms, we build on the original proofs[4]. In this section, we use the following terms:

- $\mathbb{S}$: the set of all programs, elements are s_1, s_2, s, s', etc.
- $m : \mathbb{S} \to A$: the metrics function
- $\oplus : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$: the extension function (it extends a program with another one)
- $s_1 + s_2$: simple textual concatantion of s_1 and s_2

(1)

$$\exists s_1 \in \mathbb{S} \wedge s_2 \in \mathbb{S} : s_1 \neq s_2 \implies m(s_1) \neq m(s_2)$$

It is trivially easy to find two different programs with different complexities: one example can be a more defined `catch` block that has more branches than the other.

(2)

$$\text{Let } c \in A, c \geq 0, \mathbb{S}_1 \subset \mathbb{S} | \forall s \in \mathbb{S}_1 : m(s) = c \implies |\mathbb{S}_1| \leq \infty$$

The original A-V only meets this requirement if the data connection edges are present multiple times when there are multiple uses of the data[4]. If we keep this constraint while extending the metrics, we will keep it intact, as we have defined data access for exceptions as regular variables.

(3)

$$\exists s_1 \in \mathbb{S} \wedge s_2 \in \mathbb{S} : m(s_1) = m(s_2)$$

The simplest example here would be the snippet of code that calls two different functions that can throw the same exceptions (the number of exeptions can be 0).

(4)

$$\exists s_1 \in \mathbb{S} \wedge s_2 \in \mathbb{S} : s_1 \neq s_2 \implies m(s_1) = m(s_2)$$

Two programs can meet this axiom if they, for example, differ in the way where they handle exceptions. If one function catches all possible exceptions in its body, while another one does not conatin any `catch`es, their complexity will be the same, while they greatly differ.

(5)

$$\text{Let } s \in \mathbb{S}, s_1 \in \mathbb{S} s' = \oplus(s, s_1) \implies m(s) \leq m(s')$$

Since all extensions in the original A-V increase the calculated complexity, this monotonic axiom is also true for the extended metrics.

(6)

$$\exists s_1 \in \mathbb{S} \wedge s_2 \in \mathbb{S} \wedge s' \in \mathbb{S} : m(s_1) = m(s_2) \wedge \oplus(s_1, s') = \oplus(s_2, s') \implies m(s_1) \neq m(s_2)$$

For the original A-V definition this holds, because the metrics considers data flow, thus it also depends on the environment of the program. Since we inserted the exception handling code into this data flow, it still holds for the extended definition.

(7)

$$\exists s_1 \in \mathbb{S} \wedge s_2 \in \mathbb{S} : s_1 \neq s_2 (\text{only in the order of statements}) \implies m(s_1) \neq m(s_2)$$

Using the notion of nested deepness, it is easy to find programs for this constraint. Since `catch` blocks modify nested deepness, if we consider nested `catch` blocks and the same ones, but in a linearized way, we get two programs that meet this axiom.

(8)

$$\exists s_1 \in \mathbb{S} \wedge s_2 \in \mathbb{S} : s_1 \neq s_2 (\text{only in the names of variables}) \implies m(s_1) = m(s_2)$$

Since we have not used these code items in our definitions, we trivially meet this requirement.

(9)

$$\text{Let } s \in \mathbb{S}, s_1 \in \mathbb{S} \wedge s_2 \in \mathbb{S} : s = s_1 + s_2 \implies m(s_1) \leq m(s) \wedge m(s_2) \leq m(s)$$

The consideration to prove this statement is the same as for the 6th axiom: because we consider both the nested deepness and the data flow while calculating complexity, this becomes true.

5. Evaluation

In this section we will discuss how incorporating exceptions affect the complexity of the code measured by our extended metrics. We show through a simple example that exception handling tends to decrease software complexity compared to older error handling methods. We also analyse an industrial-sized software product, first accounting for exceptions, then without them, thus we show that our extension does affect complexity.

In the following example code snippets, we demonstrate two ways of error handling. On Figure 1. an old-style return value based error handling is shown, while Figure 2. achieves the same functionality with exceptions.

Based on the previous methods for calculating the extended A-V measure for the case with exceptions is 17, while the other error handling method results in 19. This increase does not seem relevant, but considering the size of the examples, and the fact that we should also examine the result of the `calcNE` call on the 12th line, we arrive at a complexity of 23, which is a 18% increase, being a considerable amount.

Apart from synthesized examples, it is worth to excercise our newly created complexity measure on a real world example. For this purpose, we have created

```
1  private HashMap<Integer, String> errors =
2          new HashMap<Integer, String>();
3
4  int calcNE(int x) {
5          if (x == 0) {
6                  return 0;
7          }
8          if (x == 1) {
9                  return -1;
10         }
11         x--;
12         calcNE(x);
13         return x;
14 }
15
16 void clientNE() {
17         int number = 15;
18         int res = calcNE(number);
19         if (res == 0) {
20                 System.out.println(
21                 "IllegalArgumentException: " +
22                         errors.get(res));
23         }
24         if (res == -1) {
25                 System.out.println(
26                 "MyExcpetion: " + errors.get(res));
27         }
28 }
```

FIGURE 1. Java functions without exceptions

Metrics	Not measuring exceptions	Measuring exceptions	Change
LOC	279830	279830	0%
McCabe	14812	16801	13,428%
A-V	50438787	52723815	4,5303%

TABLE 1. Results of analysing Tomcat

an alayzer tool based on the Eclipse JDK that calculates McCabe and A-V metrics (both in original and extended forms) for Java code. The tested software was Tomcat[14] 6. The purpose of this analysis is to show that the extensions do increase calculated complexity. On Table 1. the end results are shown.

```
1  int calcWE(int x) throws
2          IllegalArgumentException, MyException {
3          if (x == 0) {
4                  throw new IllegalArgumentException("Zero");
5          }
6          if (x == 1) {
7                  throw new MyException("One");
8          }
9          x--;
10         calcWE(x);
11         return x;
12 }
13
14 void clientWE() {
15         int number = 15;
16         try {
17                 calcWE(number);
18         }
19         catch(Exception e) {
20                 System.out.println(e.getMessage());
21         }
22 }
```

FIGURE 2. Java functions with exceptions

The increase in cyclomatic complexity is around 13% being a significant one, and this exemplifies how widely exception handling is used nowadays. This amount is exceptionally interesting cosidering that Tomcat is more of a library, while McCabe's original deliberations were for client code. This possibly leads to the conclusion that Tomcat developers use exception handling within the framework and also to notify users about errors.

The change in the results of the A-V measure is lower than cyclomatic complexity, around 4%. The probable reason for this smaller change is that regular object-oriented data usage suppressed what the extension of the metrics increased. This is in line with our intuition, since data usage for normal control flow should be much higher than that of error handling. This also describes how exception handling is used: when there are no other options, we throw one with some information about the circumstances, but that is negligible compared to normal data usage.

6. Conclusion

In this paper we analysed how exception handling affects software complexity. Since current software metrics are unaware of error handling we extended the cyclomatic complexity and the A-V software complexity measures, and we were able to gain emprircal results. The extensions were implemented as an Eclipse plug-in and followed the ideas and thoughts behind the original McCabe and A-V metrics. The extensions have been used to analyse synthesized examples and industrial-sized software.

The first main consideration for the extended definition is registering data usage in exception handling code – be it reading or writing –, including these in the original data flow graph of A-V. We also considered the effects of throwing an exception, either if it is an explicit `throw` statement or a function call that can potentially result in exceptions being thrown. For catching exceptions, we calculated the difference of nested deepness for the throws and catches, describing the path the exception object needs to take. Uncaught exceptions are counted for as well, because they tend to put complexity pressure at the call side of functions, hence we enroll them with the complexity of their cause.

We have shown that Weyuker's axioms for complexity metrics still hold for the extended measure, assuring our intention of producing a sensible definition. Through synthesized examples, we measured the effect of various error handling methods on complexity. The short examples resulted in a significant increase of 18%. We have also analyised a real world software – Tomcat 6 – using our tool created for calculating cyclomatic complexity and A-V accounting for exceptions. The results of this analysis proved that exception handling code does have a serious effect on complexity.

As future work, we will analyse larger code bases in two different versions, one that uses exceptions and one that does not, while providing the same functionality. This should enlighten the effects of error handling methods more. The tool we have created can also be extended for other programming languages than Java to widen the set of possible candidates for our analysis.

References

[1] J.C. Cherniavksy, C. S. *On weyuker's axioms for software complexity measures.* IEE Trans. Software Engineering, vol. 17, pp.1357-1365 (1991).

[2] Louden, K. C. *Programming Languages: Principles and Practice.* Course Technology, 2002. [672] ISBN-978-1111529413

[3] McCabe, T. J. *A complexity measure.* IEE Trans. Software Engineering, SE-2(4), pp.308-320 (1976).

[4] Porkoláb, Z. *Programok Strukturális Bonyolultsági mérőszámai.* PhD thesis, Eötvös Loránd Tudományegyetem, 2002.

[5] Sutter, H. *Code complexity - part I.* http://www.gotw.ca/gotw/020.htm, September 1997. 9th May 2012.
[6] Weyuker, E. J. *Evaluating software complexity measures.* IEE Trans. Software Engineering, vol.14, pp.1357-1365 (1988).
[7] Chidamber, S.R., Kemerer, C.F., A metrics suit for object oriented design. IEEE Trans. Software Engeneering, vol.20, pp.476-498, 1994
[8] Fóthi Á. Nyéky-Gaizler J., Porkoláb Z.: The Structured Complexity of Object-Oriented Programs, Mathematical and Computer Modelling 38 pp.815-827., 2003
[9] Howatt, J.W., Baker, A.L.: Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting, The Journal of Systems and Software 10, pp.139-150, 1989
[10] Piwowarski, R.E.: A Nesting Level Complexity Measure, ACM Sigplan Notices, 17(9), pp.44-50, 1982
[11] Porkoláb, Z., Sillye, Á.: Towards a multiparadigm complexity measure, In. Proc of QAOOSE Workshop, ECOOP, Glasgow, pp.134-142, 2005
[12] Seront, G., Lopez, M., Paulus, V., Habra, N.: On the Relationship between Cyclomatic Complexity and the Degree of Object Orientation, In Proc. of QAOOSE Workshop, ECOOP, Glasgow, pp. 109-117, 2005
[13] Ádám Sipos, Norbert Pataki, Zoltán Porkoláb: On multiparadigm software complexity metrics, Pu.M.A vol. 17 (2006), No 3-4, pp. 469-482.
[14] http://tomcat.apache.org

Eötvös Loránd University
E-mail address: njeasus@ceasar.elte.hu, gsd@elte.hu