Detecting and Confining Software Errors: An Evaluation of Three Executable Assertions Based Solutions

Sébastien Perron, Mourad Badri*, Linda Badri
Software Engineering Research Laboratory, Department of Mathematics and Computer Science, University of Quebec, Trois-Rivières, 3351 Boul. des Forges, Trois-Rivières, QC G8Z 4M3, Canada.

* Corresponding author. Tel.: 819 376-5011 #3834; email: mourad.badri@uqtr.ca
Manuscript submitted March 23, 2022; accepted June 1, 2022.
doi: 10.17706/jsw.17.4.137-148

Abstract: The complexity of current software and the various dependencies between their components can affect their reliability, in particular their ability to tolerate faults. In object-oriented software, the interactions derived from such dependencies can lead to the propagation of errors. This problem can be resolved by implementing error detection and error containment mechanisms. Such techniques are used to tolerate faults. In this paper, we present three solutions where a component is reinforced in order to detect errors and limit their propagation to dependent components.

Key words: Fault tolerance, error-checking, fault injection, measurement techniques, performance attributes, software engineering.

1. Introduction

A software is composed of different components, whose behavior is defined according to their specifications, inputs and outputs that they produce. The outputs of a component are consistent with the inputs received and its internal logic. A failure occurs when the system environment observes a system output that does not conform to the software specifications [1]. An error usually originates from a latent fault located in a component. The occurrence of such errors encourages the emergence of new errors that can lead to observable failures. As a tightly coupled system increases in complexity, the potential for fault propagation (errors and failures) also increases [2].

Fault tolerance has long been deployed exclusively in high-level systems. However, faults are not exclusive to this type of systems as they occur in any system designed by humans. Some software components rely exclusively on external services, that is libraries of all kinds including recent ones such as deep learning libraries [3]. Error propagation can be mitigated by implementing protection mechanisms (error detection and error containment) in specific components. This would strengthen critical components and limit or even completely contain the propagation of their errors.

In this paper, we present three solutions to limit error propagation. These three solutions have been designed to be grafted onto existing components. They have been evaluated and compared in an empirical study using a concrete case study (JHotDraw). The results of the evaluation demonstrate the relevance of the presented solutions.

The remainder of the paper is organized as follows: Section 2 presents the problem and the solutions that were developed to solve it; Section 3 presents the three object-oriented solutions that were developed and the underlying technical principles; Section 4 presents the case study used for the evaluation of the solutions;

Section 5 presents the methodology of this evaluation; the results are provided in Section 6; a discussion follows in Section 7; Section 8 concludes the paper.

2. Problem Statement and Related Work

Protection mechanisms aim to increase the resilience of software to faults. Software resilience means "the ability of a system to consistently and reliably provide its services, especially in the face of changes, failures, and intrusions [4]". A fault tolerance mechanism is a set of features implemented to handle faults. Two steps are necessary: the errors are detected first and then they are contained. This two-step process limits the propagation of an error to other components, thus improving the robustness, availability and reliability of the software.

Classical fault tolerance relies on solutions developed according to design diversity: redundancy [5]. This solution remains limited and costly, especially since it increases the complexity of a system [6]-[8]. Current solutions are based on executable assertions [9] that allow runtime verification [10]-[15]. Many methods have been proposed to confine errors: partitioning [16], encapsulation [17]-[19] based on the Adapter pattern [20], reflection [11], [13], [21]-[23] and even the use of aspect-oriented programming [24].

The approaches developed so far have mostly been developed to provide dependability to off-the-shelf components (COTS). We have tried to design solutions that can be implemented in any critical object-oriented software (the predominant paradigm), and that are the least invasive possible (code modifications are kept to a minimum).

3. Solutions

As shown in Fig. 1, the proposed solutions are based on a similar model. A layer intercepts and filters a component's inputs and outputs. The three possible configurations are detailed in the following subsections.

Fig. 1. Basic structure of a protection mechanism.

3.1. Adapter Design Pattern

The Adapter design pattern [20], [25] is used to contain errors appropriately using an executable assertion. Just like an object adapter, one component is encapsulated in another. The code must be accessible since the component targeted by the solution application must be modified. The adapter is seen as a wrapper transforming a component into an error containment unit.

An interface is created in which the target class' method definitions are added. These methods are redefined by both the adapter class and the class being evaluated. The target class remains independent of the adapter; its visibility level remains unchanged. The adapter instantiates an object of the type of the target class and implements the methods of the target class through the newly created interface. Since the adapter now handles the target class' requests, the object instantiations of the old class for the adapter must be adapted. When a method is called on the adapter, the adapter first checks whether the input parameters are valid using an executable assertion. The check is implemented by an if statement. If the parameters are valid, the original method is called in the target class. A second check is performed on the return of a result of the method to ensure that the result is valid. When the condition is not satisfied, an exception is thrown.

3.2. Java Reflection

Java reflection is used to confine errors appropriately using an executable assertion. The reflexive process is embodied by the implementation of *java.lang.reflect. InvocationHandler*, available since JDK 1.1. The reflexive process is provided by a proxy sharing an interface with the component to be protected. The Java Reflection API is a concretization of the Proxy design pattern [20].

As for the previous solution, an interface is created in which the headers of the methods of the targeted class are added. We then create a class implementing *java.lang.reflect.InvocationHandler* (this class corresponds to the proxy, and is thus a handler of the invocations of the targeted class). When a method call is made on the targeted class, the *invoke* method in the proxy observes the invocation. In this method, we first check the name of the called method. When the method name matches that of the targeted method, the validity of the input parameters is checked by an if statement. When the parameters match, the original method is invoked on the targeted component by invoking the *invoke* method. The invocation's result is then checked to ensure the validity of the output. When the condition is not met, an exception is thrown.

3.3. Aspect-oriented Solution

Aspect weaving is used to confine errors appropriately using executable assertions. AspectJ [26] is used to compile the solution. An aspect is created containing a breakpoint at the execution of a component-targeted method (*around*). The advice is executed when the method is executed to intercept the inputs and outputs. No changes to the source code are required. Using an executable assertion, we first check that the method's inputs are valid. As for the two previous solutions, the check is implemented by an if statement. When the parameters are valid, the original method is called in the target class with proceed. A second check is performed at the end of the execution of the method, thus ensuring the validity of the result. When the condition is not satisfied, an exception is thrown.

4. Case Study

In this section, we present the case study used for the evaluation.

4.1. Introduction

The evaluation focuses on the open source software JHotDraw 7 [27]. JHotDraw is a 2D graphical drawing framework developed in Java. Its design is based on design patterns [20]. Version 7.7.0 2010 is used in this evaluation. The software contains 84,077 lines of uncommented code divided into 1,094 classes (65 interfaces). The coupling factor is about 2.01% and the average cyclomatic complexity is 2.39. This case study is used to demonstrate the feasibility of our approach and to position it with respect to those of Voas [17] and Salles et al. [11], both of whom have developed solutions applied to a software component in order to improve its reliability.

We choose, in this paper, a single component in this software to apply our solutions to, a class with the highest inherent coupling factor. The *BezierFigure* class has a FAN-IN of 42. This class allows to represent a figure with Bezier curves. The operations of the class are limited to: filling; connecting segments; adding and deleting points; setting boundaries and points; drawing on the screen; etc. This class is linked to several other classes (FAN-OUT = 15). One method is chosen in the component: *getNode* (see Fig. 2). The integer passed as a parameter makes the method easily manipulated by fault injection. The method gets a node of the curve belonging to the Bézier curve. The obtained node is cloned and returned to the client. The method is only composed of a single line of code: a call to a BezierPath method, hence the great risk of error propagation.

```
public BezierPath.Node getNode(int index) {
   return (BezierPath.Node) path.get(index).clone();
}
```

Fig. 2. Method selected for the evaluation.

4.2. Implementation of the Solutions

The executable assertions are written according to the comments written in the headers of the dependent methods (get methods in the *BezierPath* class, *get* in *java.util.ArrayList* and *checkIndex* in *java.util.Objects*). As it can be seen in Fig. 3, the index must correspond to the position of a node located on the curve, the curve corresponding to a list with variable dimensions. The runtime precondition of getNode specifies that the value of the parameterized integer must be greater than zero and less than the size of the list. Implicitly, this 32-bit signed integer is non-zero and within the bounds of its type (between -2^{31} and $2^{31} - 1$).

```
if ( !(index ≥ 0 && index < super.getNodeCount()) )
   throw new AssertionError();

BezierPath.Node result = getNode(index);

if (result == null)
   throw new AssertionError();
```

Fig. 2. Executable assertions designed for the evaluation.

The output node object is a duplicate of an existing node, so we assume that it cannot be null. We refer to the previous section for the methodology of implementing the error containment mechanisms. The experiments were run on a 3.1 GHz Intel Core i7 dual-core processor with 16 GB of memory in the IDEA 2020.1.1 development environment [28], running macOS 10.15.4. The following library and software versions were used: JRE 1.8.0_241; JUnit 4.13; AspectJ Runtime 1.9.5; Javassist 3.26; JMH 1.23.

5. Methodology

Fault injection tests fault tolerance mechanisms according to specific inputs, i.e., the faults they are designed to tolerate. This process is typically employed to assess the robustness of an application [29], [30]. In our case, faults are injected at the parameter level of the targeted method. Typically, the interface of a class is targeted by injecting randomly generated values into it. The statistics obtained are used to determine an error coverage rate. The detection coverage rate typically depends on the error detection and containment mechanism used [31]. Statistical inference is frequently used to derive meaningful reliability measures following the injection of a limited number of random faults [32], [33].

Two types of fault injections are performed: one in the method parameters and the other at the structural level. For each test carried out, the ratio between the results obtained before and after the implementation makes it possible to evaluate the robustness of the error detection mechanism. Thus, the coverage of detected errors can be determined. Others criteria are complementary to the evaluation: size of the solution in lines of code [10]; performance according to execution time [10], [14], [15], [23], [34]; processor use [15]; whether or not propagation has been observed [11], [34], [35]. These criteria will support the discussion of the results.

The first fault injection, namely *fuzz tests* [36], operates in the parameters of the designated method by

injecting random values into them. A JUnit test case performs this task. JUnit is a framework for developing and executing unit tests [37]. In our case study, a pseudorandom integer uniformly distributed between -2^{31} and $2^{31} - 1$ is generated using the *ThreadLocalRandom* singleton (the generation is isolated from the current thread) and is injected a hundred times into the parameter of the evaluated method. The number of iterations thus allows a greater consistency in the obtained results. The test case is parameterized with *@RunWith(Parameterized.class)* to be able to run the tests individually, thus improving the readability of the results. Based on the results obtained, the error detection coverage rate is determined.

A second fault injection is performed at the structural level, i.e. in the bytecode. A Java agent [38] is used to intercept the targeted class and Javassist [39] to manipulate the code. The bytecode is more easily manipulated with Javassist than with the API provided by the JDK. The agent allows the offending code to be integrated with the JVM at runtime, which is a negative value.

We use several code metrics to evaluate several quality criteria: maintainability, testability and performance. Although these metrics are secondary to the interpretation and discussion of fault injections, the contribution of these features is important: they allow us to quantify the quality of the code, especially in the case of invasive solutions. Software metrics are useful in evaluating quality attributes [40, 41, 42, 43] and crucial in decision making [44]. Moreover, the availability of such metrics to characterize solutions seems essential to us since not all solutions have the same level of maintainability, testability and do not perform in the same way.

Maintainability is quantified using the metrics of Chidamber and Kemerer [45]: CBO, WMC, RFC, LCOM, and the metrics FAN-IN and FAN-OUT. All these metrics are collected on the designated class and on the solutions with MetricsReloaded [46]. The MI maintainability index [47] is also used, which is based on the metrics of [48] and [49] and then implemented in Visual Studio [50]. The value of MI is obtained by the following formula:

$$\mathbf{MI = 171 - 5.2\ ln(V) - 0.23(VG) - 16.2\ ln(LOC)}$$

where V is Halstead's volume, VG is cyclomatic complexity, LOC is the number of lines of code.

To evaluate the performance, JMH [51] is used to compute the average execution time of the targeted method. The runtime collection is preceded by a JVM "warm-up" period during which the code is executed without taking any action. Five warm-up iterations of 10 seconds each are executed on four threads for a period of 50 seconds. This allows the machine to allocate necessary resources and compile the bytecode for the calculated iterations. The accuracy of the results depends on the number of iterations done. A high number of iterations is required, simulating in a way the conditions found in production software [52].

6. Results

We present the results of the case study evaluation in three parts: first are presented the results of fault injections to determine the robustness of the solutions (random data tests and structural tests); secondly are the code metrics; and finally, the performances of the solutions are presented in terms of average execution time.

6.1. Robustness

In JHotDraw, the 100 integers injected in the method parameter are validated by an assertion checking if the number is used by the method, in other words if the fault becomes active. Let us recall that the method is used to obtain a node of the Bézier curve. As the parameter of the method corresponds to the index of a node, its value must be between zero and the size of the array length. The index corresponds to the position in the array of *BezierPath* objects where the nodes are stored. The results observed during this fault injection with and without the solutions applied are presented in Table 1. The number of injected faults corresponds to the

proportion of invalid numbers with respect to the specifications. Of the 100 numbers injected on the software without solutions, 45% of the inputs were considered invalid. Each of these faults caused an *IndexOutOfBounds* exception to be thrown in *BezierFigure*, handled by the try-catch block of the test case. The numbers that were valid against the specification had no effect on the software. For the adapter, 43 faulty inputs were injected into the designated method. Each was detected as faulty and the *AssertionError* exception was raised. No failures or error propagation could be observed. For the proxy, 46 faulty numbers were injected and all were handled. 40 faulty numbers were injected in the aspect solution and each of them could be detected by the advice and an exception was raised.

Table 1. Distribution of Faults Injected in the Method Parameters of *getNode*

	Injected	Detected	Propagated
Without solution	45	0	45
Adapter	43	43	0
Proxy	46	46	0
Aspect	40	40	0

Since the anti-propagation capabilities of the solutions could not be tested on getNode, the same evaluation is repeated, but on a separate class. A component is again chosen according to different criteria this time. This is done by examining the dependent classes of BezierFigure. The BezierControlPointHandle class is chosen given the ease with which the index set in the constructor is handled in one of its methods. The class represents a figure's control point's manipulation handle (extends *AbstractHandle*, which implements *Handle*). As always, only one method is targeted: *getBezierNode* (the source code is presented in Fig. 4). When this class is instantiated, the index of a node is passed in the parameters. This same index is used by the previously evaluated method. Although the method does not accept any parameters, the index passed in the constructor makes fault injection possible. The assertion must therefore detect whether the index is greater than zero or not since the method does not check if the index is negative or out of bounds (array length minus 1).

```
@Nullable
public BezierPath.Node getBezierNode() {
   return getBezierFigure().getNodeCount() > index ?
     getBezierFigure().getNode(index) : null;
}
```

Fig. 4. Selected method for evaluating error propagation.

The results observed during this fault injection with and without the solutions are presented in Table 2. Of the 42 faulty numbers generated and injected into the software without solutions, none could be detected. All of them caused an *IndexOutOfBounds* exception to be thrown in *BezierFigure* and were handled by the test case try-catch block. Each error was propagated in the following sequence: when invoking *getBezierNode* method, the index passed in the constructor and then assigned to a class variable was passed to *getNode* in *BezierFigure*; the faults activated in that class and created errors where an exception was thrown and a failure ensued. This phenomenon has not been observed when the adapter was implemented, where 69 injected faults were detected and properly contained. The situation is identical for the proxy and the aspect, which respectively detected and successfully contained 68 and 70 faults.

For the fault injection performed at the bytecode level, the injected fault assigns a negative value to the

index (−1). As expected, the fault was not detected or contained in the software without a solution. The fault caused an *IndexOutOfBoundsException* to be thrown in *BezierFigure*, handled by the test case try-catch block. The fault injected into the adapter was correctly detected and thus caused the *AssertionError* exception to be thrown, again handled by the test case. The same phenomenon was observed in the proxy in an identical sequence of events. For the aspect solution, the fault was detected by the advice and caused the *AssertionError* exception to be thrown.

Table 2. Distribution of Faults Injected in the Method Parameters of *Get BezierNode*.

	Injected	Detected	Propagated
Without solution	42	0	42
Adapter	69	69	0
Proxy	68	68	0
Aspect	70	70	0

6.2. Maintainability and testability

Table 3 shows the code metrics for the *BezierControlPointHandle* class of JHotDraw. The results do not include the interface metrics since they have, for the most part, no object correspondent. For the FAN-OUT metric, only dependencies in the software are considered (external dependencies are excluded).

Table 3. JHotDraw Source Code Metrics

	MAINTAINABILITY						TESTABILITY	
	CBO	FAN-IN	WMC	RFC	LCOM	MI	FAN-OUT	RFC
Adapter	8	4	5	9	1	95.79	4	9
Proxy	6	3	7	12	1	86.72	4	12

Maintainability is considered in terms of complexity and coupling. For dependencies: the FAN-IN index is 4 for the adapter and 3 for the proxy; the coupling between objects (CBO) is 8 for the adapter and 6 for the proxy. The WMC complexity index is 5 for the adapter and 7 for the proxy. The RFC complexity metric for JHotDraw is 9 for the adapter and 12 for the proxy. The solution is perfectly cohesive, i.e. LCOM = 0. The adapter and the proxy are highly maintainable since their index, 96 and 87 respectively, are above the 85 threshold. The testability is quantified by the number of efferent dependencies (FAN-IN) and by RFC. The number of dependencies of the adapter and the proxy is 4.

6.3. Performance

The average execution times for the *getBezierNode* method in *BezierControlPointHandle* are summarized in Table 4. The measurements are expressed in nanoseconds. For the adapter, the average processing time for the designated method is the lowest of the three solutions, at about 42 ns per operation (ns/op). These measurements are similar to those where no solution is implemented. The proxy has a slightly higher execution time than the adapter, of 57 ns/op with a 2.7 margin of error (half the adapter). With a score of 100.6 ns/op, the aspect admits a 172% increase in processing time, with a 16.7 margin of error (three times that of the adapter).

Table 4. Average execution time of *getBezierNode*

	avg (ns)	median (ns)
Without solution	37.001	2.072
Adapter	42.354	5.360
Proxy	57.113	2.736

Aspect	100.584	16.742

7. Discussion

The results indicated that the solutions were all able to prevent the propagation of errors caused by the two fault injections. The goal of the evaluation was to assess the solutions to determine the effectiveness of executable assertions as a means of error detection and the effectiveness of three containment methods. The evaluation was limited to one particular class, where a single method was selected.

When injecting random data, all solutions were able to prevent the introduction of invalid inputs into the targeted method. When assertions are designed strictly according to the software specifications, they are an effective means of error detection for the evaluated software. Assertions act as a semantic data integrity checker, a type of verification widely used in fault tolerant designs [53].

The method inputs and outputs were checked. All errors could be detected. Checking the outputs was not useful since the errors were caused before a result was even produced. Thus, during the evaluation of the *BezierFigure* class, no propagation could be observed. The method selection criteria in the methodology should be revised accordingly in order to consider a method that can cause error propagation. A strongly coupled method would have facilitated the evaluation. The solutions therefore prevent error propagation. The same results were observed when injecting faults at the bytecode level: faults were detected and adequately contained by the three containment methods.

Only object-oriented metrics have been used to quantify solutions since no metric can adequately define an aspect. Although maintainability metrics have been developed for the aspect paradigm [54], these metrics neglect the characteristics specific to the aspect language, often adapted from object-oriented [55]. For this reason, no metric has been used to quantify aspect solution complexity and coupling. The CBO metric is at near-equality in both object solutions, however the adapter is slightly more coupled than the proxy (+33%). The afferent coupling (FAN-IN) does not reveal anything special; we only note that the value for the adapter is slightly higher. The value for efferent coupling (FAN-OUT) is identical for both solutions. The LCOM metric indicates zero in both cases, meaning that the solutions are perfectly cohesive. Finally, the maintainability index is above the high maintainability threshold [47], which means that both the adapter and the proxy are equally maintainable.

The performance of the solutions was evaluated according to the average processing time per operation. The measurements range from 42 to 100 nanoseconds. The adapter has an increase in processing time of about 14.47%. The increase is more significant in the other two solutions, 54.36% for the proxy and 171.84% for the aspect solution. In the first two cases, the degradation of the software performance seems to us negligible for the observed robustness gain. However, the Java reflection library and the AspectJ weaver induce a more significant increase in execution time. The adapter is the least expensive solution, making it an ideal solution when we want the least possible degradation in performance. Note however that the sample size does not allow a generalization of these results. Thus, it could be interesting to increase the number of experiments to obtain a large enough sample to assume a good performance of solutions on object-oriented software. Regarding testability, the RFC metric being in direct correlation with the metric (DNOTC) indicating the size of a test suite [56, 57] indicates that more test cases will have to be written to test the proxy than for the adapter. The FAN-OUT and RFC metrics are good predictors of the size of the test suite [56]. The adapter has the lower testability of the two solutions. This solution therefore requires more understanding from developers.

The use of an adapter as a method of containment and executable assertions as a means of error detection was proposed for COTS [17] and subsequently as a design pattern [25]. The design of the adapter was

originally done by the authors of Design Patterns [20], whose object variant was initially considered here. The novelty of our study with respect to these two studies will have been, among other things, to evaluate this containment mechanism as a cohesive unit allowing to limit the propagation of errors.

The proxy is also loosely coupled with the software classes for the case study used. Weak coupling is desirable since we want a future modification to have the minimum risk of affecting another component. The use of reflection is a limiting factor in its transfer to other platforms, although Java is not the only language with an introspection library. A reflective object cannot be tested in the same way as an adapter, adding a level of complexity to the testing phase. For this solution, it was necessary to adapt the definition of the *BezierFigure* and *BezierControlPointHandle* classes in order to implement a class. However, the body of these classes was not altered. Executable assertions could therefore introduce new faults or errors (use of an uninitialized variable, algorithmic errors, infinite loops added during instrumentation, out-of-bounds array reference, etc.), given our limited knowledge of the characteristics of the software being evaluated. Assertions are effective when the nature of the errors to be detected is known.

The aspect-oriented process is considered similar to the one of the proxy. No difficulties were encountered during the implementation of the solution, unlike the other two. Unlike the other two solutions, this one could not be adequately tested, as no tool is able to test an aspect's pointcuts [58]. Moreover, the type of advice used in the solution implies a significant increase in processing time [59].

8. Conclusion

The complexity of software and the plurality of its inter-component dependencies can affect its reliability, in particular its ability to tolerate faults. In object-oriented software, such dependencies imply a coupling that can, in fact, lead to error propagation. This can be solved by implementing error detection and containment solutions. In the first step, we presented three solutions whose goal was to limit the propagation of errors. These solutions were successively implemented and evaluated in a Java object-oriented software. The proposed solutions are based on the use of executable assertions as a means of error detection and on three distinct means of error containment: the Adapter design pattern, Java reflexivity and aspect weaving. Two types of fault injections were used: the first test determined the robustness of the solutions by injecting random numerical values into the parameters of a method; the second test was performed at the bytecode level, where a variable was instrumented. According to the results, all solutions were able to properly detect the injected faults and prevented error propagation. When the assertions are developed according to the specifications, these solutions are effective in detecting and preventing fault propagation in the data of an object-oriented software.

Conflict of Interest

The authors declare no conflict of interest.

Authors Contributions

Sébastien Perron: Methodology, Investigation, Solutions Design and Implementation, Empirical Study, Results Analysis, Writing - Original Draft, Writing - Review & Editing, Visualization

Mourad Badri: Project Definition, Conceptualization, Methodology, Solutions Analysis, Empirical Study, Writing - Review & Editing, Supervision, Project administration

Linda Badri: Methodology, Empirical Study, Writing - Review & Editing, Co-Supervision

References

[1] Laprie, J. C. (1995). Dependable computing and fault tolerance: concepts and terminology. *Proceedings*

of the International Symposium on Fault-Tolerant Computing.

[2] Perrow, C. (1984). *Normal Accidents: Living with High-Risk Technologies.* Basic Books.

[3] Xiao, Q., Li, K., Zhang, D., & Xu, W. (2018). Security risks in deep learning implementations. *Proceedings of the 2018 IEEE Security and Privacy Workshops* (pp. 123-128).

[4] Laprie, J. C. (2008). From dependability to resilience. *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

[5] Avizienis, A., & Kelly, J. P. J. (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer, 17*(8), 67-80.

[6] Anderson, T., Barrett, P. A., Halliwell, D. N., & Moulding, M. R. (1985). Software fault tolerance: An evaluation. *IEEE Transactions on Software Engineering, SE-11*(*12*), 1502-1510.

[7] Carzaniga, A., Gorla, A., & Pezzè, M. (2009). Handling software faults with redundancy. *Architecting Dependable Systems VI* (pp. 148-171). Springer.

[8] Downer, J. (2009). *When Failure is An Option: Redundancy, Reliability and Regulation in Complex Technical Systems.*

[9] Mahmood, A., Andrews, D. M., & McCluskey, E. J. (1984). *Executable Assertions and Flight Software* (Report No. 84-258). Stanford, CA: Center for Reliable Computing, Stanford University.

[10] Leveson, N. G., Cha, S. S., Knight, J. C., & Shimeall, T. J. (1990). The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering, 16(4)*, 432-443.

[11] Salles, F., Rodriguez, M., Fabre, J.-C., & Arlat, J. (1999). Metakernels and fault containment wrappers. *Proceedings of the International Symposium on Fault-Tolerant Computing*.

[12] Hiller, M. (2000). Executable assertions for detecting data errors in embedded control systems. *International Conference on Dependable Systems and Networks* (DSN 2000).

[13] Popov, P., Riddle, S., Romanovsky, A., & Strigini, L. (2001). On systematic design of protectors for employing OTS items. *Proceedings of the EUROMICRO Conference 2001*.

[14] Rodríguez, M., Fabre, J.-C., & Arlat, J. (2002). Wrapping real-time systems from temporal logic specifications. *Dependable Computing EDCC-4*, 253-270.

[15] Afonso, F., Silva, C., Brito, N., Montenegro, S., & Tavares, A. (2008). Aspect-oriented fault tolerance for real-time embedded systems. *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software.*

[16] Hanmer, R. S. (2007). *Patterns for Fault Tolerant Software.* Wiley.

[17] Voas, J. M. (1998). Certifying off-the-shelf software components. *Computer, 31(6)*, 53-59.

[18] White, I. (2000, April). Wrapping the COTS dilemma. *Commercial Off-the-shelf Products in Defence Applications "The Ruthless Pursuit of COTS" IST Symposium*, Brussels, Belgium.

[19] Anderson, T., Feng, M., Riddle, S., & Romanovsky, A. (2003). *Investigative Case Study: Protective Wrapping of OTS Items in Simulated Environments* (Report No. CS-TR-821). Newcastle upon Tyne: University of Newcastle upon Tyne.

[20] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

[21] Maes, P. (1987). Concepts and experiments in computational reflection. *OOPSLA '87*, Orlando, Florida, 147–155.

[22] Fabre, J.-C., Nicomette, V., Perennou, T., Stroud, R. J., & Zhixue, W. (1995). Implementing fault tolerant applications using reflective object-oriented programming. *Proceedings of the International Symposium on Fault-Tolerant Computing*.

[23] Fraser, T., Badger, L., & Feldman, M. (1999). Hardening COTS software with generic software wrappers. *Proceedings of the IEEE Symposium on Security and Privacy*.

[24] Shah, V., & Hill, F. (2003). An aspect-oriented security framework. *Proceedings of the DARPA Information Survivability Conference and Exposition*.
[25] Saridakis, T. (2003). Design patterns for fault containment. *Proceedings of the EuroPLoP Conference*.
[26] Eclipse Foundation. (2019). Retrieved from: https://www.eclipse.org/aspectj/
[27] Randelshofer, W. (2010). Retrieved from: https://sourceforge.net/projects/jhotdraw/
[28] Intellij. (2020). Retrieved from: https://www.jetbrains.com/idea/
[29] Voas, J., &McGraw, G. (1997). *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons.
[30] Feinbube, L., Pirl, L., & Polze, A. (2018). Software fault injection: A practical perspective. In F. P. García Márquez and M. Papaelias (eds.), *Dependability Engineering* (pp. 47-60). IntechOpen.
[31] Constantinescu, C. (1994). Estimation of coverage probabilities for dependability validation of fault-tolerant computing systems. *Proceedings of the COMPASS'94 - 1994 IEEE 9th Annual Conference on Computer Assurance, Gaithersburg*.
[32] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C., & Powell, D. (1990). Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering, 16*(*2*), 166-182.
[33] Powell, D., Martins, E., Arlat, J., & Crouzet, Y. (1993). Estimators for fault tolerance coverage evaluation. *Proceedings of the FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*.
[34] Arlat, J., Fabre, J.-C., Rodríguez, M., & Salles, F. (2002). Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers, 51*(*2*), 138-163.
[35] Fabre, J.-C., Salles, F., Moreno, M. R., & Arlat, J. (1999). Assessment of COTS microkernels by fault injection. *Dependable Computing for Critical Applications.*
[36] Miller, B., Fredriksen, L., & So, B. (1989). An empirical study of the reliability of UNIX utilities. *Communications of the ACM, 33*(*12*), 33-44.
[37] JUnit. (n. d.). JUnit (4.13). Retrieved from: https://junit.org/junit4/
[38] Oracle. (n. d.). Java™ Platform Standard Ed. 7. java.lang.instrument package. Retrieved from: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html
[39] Chiba, S. (2019). Javassist (3.25.0.GA). Retrieved from: https://www.javassist.org
[40] Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22*(*10*), 751-761.
[41] Fenton, N., & Pfleeger, S. (1997). *Software Metrics: A Rigorous and Practical Approach* (2nd edition). PWS Publishing Co.
[42] Pressman, R. (2014). *Software Engineering: A Practitioner's Approach* (8th edition). McGraw-Hill.
[43] Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson.
[44] Harrison, W. (1994). Software measurement: A decision-process approach. *Advances in Computers, 39*, 51-105.
[45] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering, 20*(*6*), 476-493.
[46] Leijdekkers, B. (2020). Retrieved from: https://plugins.jetbrains.com/plugin/93-metricsreloaded
[47] Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer, 27*(*8*), 44-49.
[48] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2*(*4*), 308-320.
[49] Halstead, M. H. (1977). *Elements of Software Science*. Elsevier.
[50] Naboulsi, Z. (2011). *Code Metrics —– Maintainability Index.* Retrieved from:

https://blogs.msdn.microsoft.com/zainnab/2011/05/26/code-metrics-maintainability-index/
[51] OpenJDK. (2020). Java Microbenchmark Harness (JMH). Retrieved from: https://github.com/openjdk/jmh
[52] Antony, A. (2021). *Understanding Java Microbenchmark Harness or JMH Tool.* Retrieved from: Https://Medium.Com/Javarevisited/Understanding-Java-Microbenchmark-Harness-Or-Jmh-Tool-5b9b90ccbe8d
[53] Dubrova, E. (2013). *Fault-Tolerant Design.* Springer.
[54] Ceccato, M., & Tonella, P. (2004). Measuring the effects of software aspectization. *1st Workshop on Aspect Reverse Engineering* (WARE 2004), Delft, The Netherlands. Retrieved from: https://selab.fbk.eu/ceccato/papers/2004/ware2004.pdf
[55] Burrows, R., Garcia, A., & Taïani, F. (2010). Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies. *Evaluation of novel Approaches to Software Engineering* (pp 277-290). Springer.
[56] Bruntink, M. (2003). *Testability of object-oriented systems: a metrics-based approach* (Master thesis, Universiteit van Amsterdam). https://homepages.cwi.nl/~paulk/theses/Bruntink.pdf
[57] Bruntink, M. and van Deursen, A. (2004). Predicting class testability using object-oriented metrics. *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, Chicago, IL, 136-145.
[58] Delamare, R., Baudry, B., Ghosh, S., Gupta, S., & Le, T. Y. (2011). An approach for testing pointcut descriptors in AspectJ. *Software Testing, Verification & Reliability, 21*(3), 215-239.
[59] Šuta, E., Martoš, I., & Vranić, V. (2015). Usability of aspectj from the performance perspective. *Proceedings of the IEEE 1st International Workshop on Consumer Electronics*.

Sébastien Perron is a graduate student. He received his master in applied mathematics and computer science from the Department of Mathematics and Computer Science of the University of Québec at Trois-Rivières, Canada. His main research fields are software engineering, object-oriented development, and software quality assurance.

Mourad Badri is a full professor of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières (Canada). He holds a PhD in computer science (software engineering) from the National Institute of Applied Sciences in Lyon, France. His main areas of interest include object and aspect-oriented software engineering, software quality attributes, software testing, refactoring, software evolution and application of machine learning in software engineering.

Linda Badri is a full professor of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières (Canada). She holds a PhD in computer science (software engineering) from the National Institute of Applied Sciences in Lyon, France. His main areas of interest include object-oriented software engineering, Software development and maintenance, Software quality assurance and refactoring.