Returning Extended Information about Errors Enclosed in the Result Object instead of Anticipated Exceptions in Java

Abstract

During the program execution, various exceptional conditions can occur, which can affect the program's normal behavior. These exceptional situations can be divided into two main categories: anticipated and unanticipated exceptions. Typically, if an expected exceptional condition occurs, the specific details of the exception are less important than the fact that this special kind of exception occurred. In this scenario, using the standard exception mechanism provided by the programming language to handle expected errors might be overused or even incorrect and lead to increasing program complexity and time overhead. In the paper the authors overview the methods used for error and exception handling in various programming languages, examine the advantages and disadvantages of these methods. Also, the solution for Java programming language is proposed which is based on returning the result object composed of the value and notification which eliminates a number of disadvantages associated with the exception handling. Applying this method in scenarios such as validation of user-entered values or processing data received from external sources can reduce the overhead of handling exceptions in a standard way, uniformly providing the necessary information about the possible corrections, decreasing the number of iterations for checking the correctness of the data. Also, some ideas of further improvements are proposed which are reserved for future works.

Keywords

Exception, Exception mechanisms, Exception handling, Notification, Error handling, Java

1. Types of errors

During the execution of the program, various exceptional situations may arise that are not anticipated by the specification of the program state. Exceptional situations can occur for various reasons, including logical errors in the program, errors in the input data, errors in the program environment, and so on. In some cases, the raising of an exception is fatal for the program, since at the runtime there is no way to return to the normal state, correct the error, and continue the program. In this case, the only option is the abnormal termination of the program with a possible message about the occurrence of an exception.

However, in many cases, error conditions can correct the situation, and either redo the actions that led to an exceptional situation or ignore the result of erroneous actions at all, if its absence does not affect the further execution of the program. For such cases, many modern programming languages use exception handling mechanisms, which can be described as follows. Programs consist of components that request other components to perform specific actions. The called component returns the result to the calling component. During the normal execution of the program, it can be viewed as a sequence of requests and corresponding responses. When an exception happens, the component throws an exception object, holding information about the exception, interrupts its work, and returns the exception object to the called component, signalling that execution did not complete correctly. If the calling component has all the necessary information to correct the error, it transfers control to the exception handler. The exception handler performs the necessary actions, after which the program returns to normal operation. If the required handler is absent, then the exception object is propagated via the invocation hierarchy to find handlers for the exception. If the exception reaches the main module, and the handler is still absent, the error is considered fatal and the program terminates abnormally.

This mechanism is quite universal and implies the ability to handle even those exceptions that were not anticipated by the program developers. In addition, it does not provide the ability to ignore exceptional situations, preventing the program from continuing execution when an error occurs. However, the convenience of this method of handling exceptions leads to the fact that it is often used to handle both non-standard and standard situations, including those that are provided by specification with a handler to a situation that has arisen.

As already mentioned, exceptional situations can be fatal and correctable. Further, we will consider only correctable situations, since it makes no sense to handle fatal errors - they must terminate the program abnormally. Correctable exceptions can also be divided into expected and unexpected. Errors that are related to the general logic of the program are expected errors. For example, it might be an invalid user input value. All other errors not related to business logic will be considered unexpected. It is important to note that errors resulting from contract violation are unexpected (for example, when a function is called with an argument for which a precondition is

Proceedings of the 13th Majorov International Conference on Software Engineering and Computer Systems,
December 1-3, 2021, Online & Saint Petersburg, Russia

not met, an unexpected error should occur, since the handling of such a situation cannot be determined by the business logic of the application) The standard exception handling mechanism is also well suited for unexpected errors, as it was originally intended to handle situations that are not covered by the specification.

Given that expected and unexpected errors are two different classes, each with different causes and potentially different handling, it is natural to use different mechanisms to handle these two types of exceptions. In particular, it makes sense to handle unexpected errors in order to maintain program invariants when errors occur [1]. At the same time, expected exceptions directly follow from the business logic of the application, and, most often, are associated with the need to validate the input data. For such situations, the mechanism for interrupting the program is not always preferable. In this regard, this work will mainly consider expected exceptional situations, that is, those that are part of the overall logic of the program. In this case we can easily return the appropriate result which often is more suitable.

There are several options for using the error handling mechanism in various object-oriented programming languages, differing in the presentation of the exception object, the obligation to declare thrown exceptions, the classification of exceptions, the binding of exception handlers, the ability to repeat the error action, and the presence of control over the handling of exceptions [2]. This article is mainly devoted to the Java language, in which exceptions are divided into checkable and unchecked, the first of which are of greatest interest, since they are most often used to handle situations that can be provided in the specification and logic of the program.

2. Methods of handling exceptions in programming languages

In the early days of computing, failures during the execution of a program always caused the program to terminate abnormally. For some types of errors, this behavior is still the best way to handle the error. However, there are many types of errors that are not fatal, and appropriate actions can be taken to correct such errors, after that the program execution can continue. Various ways of reacting to occurring non-standard conditions are used to handle the errors. For example, in the FORTRAN programming language, labels can be defined to jump to the appropriate block of code in case of an error. A similar approach is used in some contemporary programming languages, but instead of providing jump addresses, handlers for different types of errors are directly passed to the called function.

In structured programming languages, a simple transfer of execution to another code block would lead to a violation of the consistency of the program structure. One of the methods for handling non-standard situations arising during a program operation is using a predefined special value as a result, namely status or error code, returned from a function or procedure. If the function does not need to return a value, it can return only the status code indicating the presence or absence of an error. If the function does need to return a value, then the presence of an error can be indicated by a value that differs from the usual one in its type or range. Another way to signal an occurrence of an error is to assign a value to a dedicated global variable named error flag. Additionally, if supported by the type system, it is possible to return a pair of values, both the result and the completion status in a single object. All the above-stated approaches do not require any special syntactic language constructs. They have been used in programming languages lacking special exception handling mechanisms, for example, C, Rust or Go. The advantage of this approach is its simplicity and predictable speed of code execution.

However, these methods can pass the error code only by returning from the called function to the caller function, which leads to redundant code especially in cases of deeply nested calls. Also, the disadvantage of returning the error code is the lack of control, because the called function is unable to force the calling function to check for an error. If the error code is not checked and its value is ignored this can lead to program inconsistency. Another drawback is the absence of detailed information about the error and its cause, and the error codes are not unified and require special ways to provide compatibility between program components.

As a result, a method for handling exceptions in structured language was proposed [3], which is currently supported syntactically in most modern programming languages. This method is based on declaring a special block that defines the scope of exception handling. A block of code containing a call to a function that potentially can throw an exception is accompanied by handlers for each type of exception. Optionally, there can be a special continuation block that is executed only when the function call ends normally or a special finalization block that is executed on both normal and abnormal termination of a function. When an exceptional situation occurs, a special exception object is generated, which contains all the necessary information about the exception. Then the normal execution of the called function is interrupted, an emergency return occurs unwinding the call stack until a handler for the matching type of exception is found. If there is no such handler, the whole program will terminate abnormally.

The benefits of this method are the following. First, the calling function is informed about the error automatically. Second, the method can be dynamically adjusted to various types of exceptions including those that may be

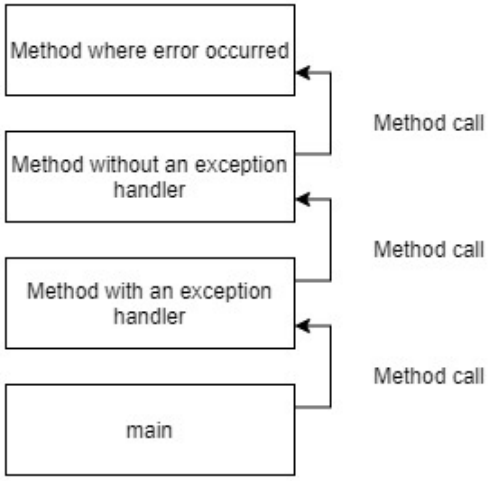

Figure 1: Call stack

required in the future. Third, the exceptions are handled at the level where there is enough information to make a decision on possible actions to correct the situation. In addition, this method allows the compiler to check the presence of an exception handler during compilation of the source code, thereby preventing the possibility to ignore the potentially erroneous situations.

The disadvantages of this approach are the relatively large overhead of handling exceptions compared to error codes, the unpredictability of control flow during static code analysis, and the inability to handle more than one error at a time.

3. Exceptions in Java

Exceptions in Java catch the stack trace at the moment of initialization. They are created as regular objects and are called exception objects that contain information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

To handle exceptions in Java, the traditional method of processing for modern languages is usually used using a special syntactic construction (the try-catch-finally block). Despite some innovations as the language develops, this mechanism is not without drawbacks [4], and in many cases, it's not quite correct usage causes additional errors to be introduced into the program [5]

The runtime system searches the call stack for an exception handler. The search begins with the method in which the error occurred and proceeds through the call

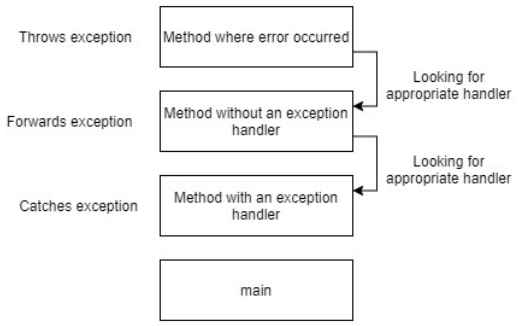

Figure 2: Searching the call stack for the exception handler

stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler as shown in figure 3

If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates. This behavior is useful when debugging a program, but these actions have their costs.

A study was conducted on the execution time of one operation with throwing an exception and without the corresponding operation using JMH library. The following configuration was used to conduct the study: JDK 15.0.1, OpenJDK 64-Bit, 15.0.1+9-18, AMD Ryzen 3600 (4.2 GHz)

```
@Benchmark
public int withException() {
    try {
        return actionWithException();
    } catch (IllegalArgumentException e) {
        return value;
    }
}
@Benchmark
public int withoutException() {
    return action();
}
private int action() {return value;}
private int actionWithException() {
    throw new IllegalArgumentException();
}
```

Listing 1: Simple Exception Benchmark

Table 1 shows the execution time of one operation in nanoseconds (Score) for the corresponding benchmark.

Table 1
Comparison benchmarks “with” and “without” exception with different call stack depth

Benchmark	Score	Error	Units
withException	1071,466	± 24,251	ns/op
withoutException	3,680	± 0,020	ns/op

As can be seen from the results obtained, the method using the exception mechanism (withException) is much slower than the method without using it (withoutException)

Since the exception mechanism involves iterating through the call stack, then with an increase in its depth, the speed of the operation decreases even more. To demonstrate this behavior, two simple methods were created, which, depending on the depth, call themselves recursively. The code of the corresponding methods is given in listing 2

```
@Benchmark
public int withException() {
    try {
        return actionWithException();
    } catch (IllegalArgumentException e) {
        return value;
    }
}
@Benchmark
public int withoutException() {
    return action();
}
private int action() {return value;}
private int actionWithException() {
    throw new IllegalArgumentException();
}
```

Listing 2: Methods with controlled depth in the call stack

As can be seen from the listing 2, the withException method throws an exception at the end, and the withoutException method returns -1 code. Benchmarks were created for the method with an exception together with the processing in the try-catch block and the method without using an exception. The results presented in the table 2

As can be seen from table 2 the method is executed slowly due to the deep call stack and with the value of the depth variable equal to 150, we get a relative difference of 127 times between execution without exception and with it. Thus, exceptions significantly reduce performance, especially when the stack depth increases, which is confirmed by other authors [6]

Sometimes detailed error information is not as important as the fact that the expected error occurred. In this case, using the exception mechanism may be unnecessary or even incorrect. It is also sometimes difficult to ”catch” these exceptions, which can cause so-called EH-bugs [5]. The catch-block may be placed in the wrong place or not placed at all, an exception may be thrown that should not have been thrown, etc.

In such cases, it is better to have an approach that would eliminate the disadvantages of using the exception mechanism in places where detailed information about the error or catch block is not required.

4. Error handling via Notifications

As proposed in the article [7] it is worth replacing exception handling with a special object called Notification which in certain cases can be more effective, especially when dealing with expected errors. The Notification object collects information about multiple errors at once and is used to return this information to a calling method or function, where the information could be extracted and used for correction. Fowler proposed to use his solution in scenarios when a presentation layer captures data from the user and passes it to the domain layer for validation. The domain layer makes the necessary checks and if any of them fail informs the presentation layer about errors.

However, the Notification pattern proposed by Fowler has some minor issues. For example, the information about the error is represented as a string, which is suitable for simple cases where the string can be used directly. But this method is not generalized enough, because the user cannot add his own object for error description. In other words, a Notification object is like an error flag containing some additional data.

Roughly speaking, the method should return not the value itself, but the wrapper. Java has also the Optional class, which wraps the object and provides the methods which can handle null values in a way eliminating exception throwing. But Optional wrapper only can deal with NullPointerException and doesn’t support multiple values.

A similar way of error handling was introduced in the Rust language using the Result type, which was defined in the standard library of the language (Listing 3). Depending on the result, one or another action could be performed.

Table 2
Comparison benchmarks “with” and “without” exception with different call stack depth

Benchmak	(depth)	Mode	Cnt	Score	Error	Units
DepthExceptionBenchmark.benchWithException	0	avgt	25	1078,796	± 18,343	ns/op
DepthExceptionBenchmark.benchWithException	5	avgt	25	1477,162	± 12,634	ns/op
DepthExceptionBenchmark.benchWithException	10	avgt	25	1982,386	± 13,011	ns/op
DepthExceptionBenchmark.benchWithException	20	avgt	25	3105,875	± 34,954	ns/op
DepthExceptionBenchmark.benchWithException	50	avgt	25	5860,825	± 45,027	ns/op
DepthExceptionBenchmark.benchWithException	100	avgt	25	10852,055	± 72,008	ns/op
DepthExceptionBenchmark.benchWithException	150	avgt	25	15649,591	± 129,355	ns/op
DepthExceptionBenchmark.benchWithoutException	0	avgt	25	3,975	± 0,018	ns/op
DepthExceptionBenchmark.benchWithoutException	5	avgt	25	8,726	± 0,052	ns/op
DepthExceptionBenchmark.benchWithoutException	10	avgt	25	11,631	± 0,084	ns/op
DepthExceptionBenchmark.benchWithoutException	20	avgt	25	17,850	± 0,340	ns/op
DepthExceptionBenchmark.benchWithoutException	50	avgt	25	38,376	± 0,382	ns/op
DepthExceptionBenchmark.benchWithoutException	100	avgt	25	82,638	± 0,776	ns/op
DepthExceptionBenchmark.benchWithoutException	150	avgt	25	123,894	± 1,565	ns/op

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Listing 3: Result definition in the standard Rust library

But this method has a cumbersome syntax using the unwrap function presented in Listing 4

```
impl<T, E: ::std::fmt::Debug> Result<T, E> {
 fn unwrap(self) -> T {
  match self {
   Result::Ok(val) => val,
   Result::Err(err) =>
    panic!("called `Result::unwrap()` on an
 ↪  `Err` value: {:?}", err),
  }
 }
}
```

Listing 4: Result unwrapping example

5. Approach for working with notifications in Java

When considering the approach to work with errors from the Rust language and Fowler’s approach to work with notifications, a proposal arose to take advantage of the Java language and return not only the error flag, but also additional information that can be used later for processing, or with its help change the return value depending on the error.

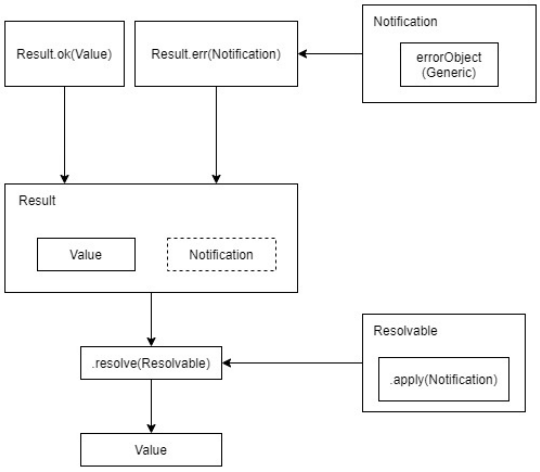

Figure 3: General structure of the program

The NoTex (Not Exception) library has been developed to implement the described approach. In NotEx the wrapper is the Result class, which contains the return value, as well as a notification if an error occurred. The general structure is shown in Figure 5

Result may not have notifications if there was no error. If there was one, then the wrapper should contain a Notification, which will then be processed in the resolve method, which accepts the Resolvable functional interface.

```
T apply(Notification<S> notification)
```

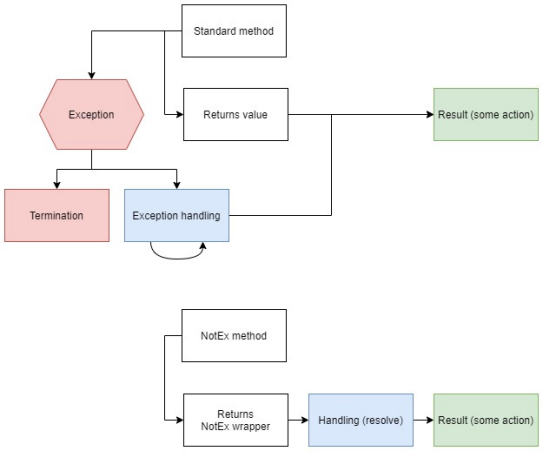

Figure 4: Comparison of the NotEx and exception mechanism in Java

The resolve method generally allows you to apply one or another action depending on the presence of an error (notification)

```
<R> R resolve(Function<T, R> function,
 ↪  Resolvable<S, R> resolvable)
```

"function.apply" is applied if Result does not contain an error and "resolvable.apply" if Result contains an error and the notification needs to be processed.

Result allows the user to write linear data processing/validation. Example is in Listing 5

```
Boolean value = Result.of("Hey!!")
        .resolveFrom(String::length)
        .apply(this::assertOdd)
        .apply(this::assertTrue)
        .resolve(n-> {
            log.info(n.getMessage());
            return false;
        });
```

Listing 5: Example of linear execution using NotEx

The conciseness of the code also increases, since the use of language syntax is not required as shown in listing 6.

Table 3
Comparison of NotEx and exception throughput

Benchmark	Score	Error	Units
benchException	958048	± 10285	ops/s
benchNotEx	274030504	± 8293934	ops/s

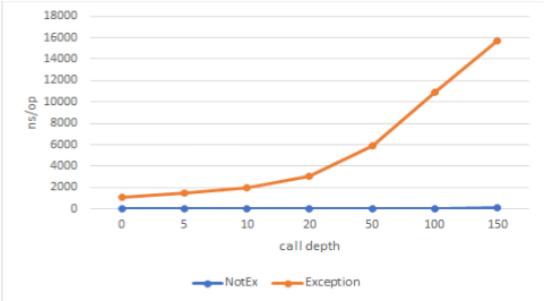

Figure 5: Graph comparing exceptions and NotEx with nested calls

```
public Integer benchException() {
    Integer value;
    try {
        value = withException();
    } catch (ArithmeticException exception) {
        value = valueIfExceptionOccurs();
    }
    return value;
}
public Integer benchNotEx() {
    return withNotEx()
        .resolve(notification ->
         ↪  valueIfExceptionOccurs());
}
```

Listing 6: Benchmarks source code

Both methods in listing 6 do the same thing, but the second one looks clearer, without cumbersome language syntax. Moreover, it works faster due to the fact that it does not use exceptions as shown in Table 3

We get a gain in speed of 286 times

This becomes even more important when throwing an exception from a nested call when the call stack grows

The results are shown in Table 4 and graphically in Figure 5. When the call stack increases, the execution time of the exception operation increases significantly faster compared to NoTex.

The benchmark was carried out with a recurrent call of two methods in Listing 7

Table 4
Comparison of exceptions and NotEx in nested calls

Benchmak	(depth)	Mode	Cnt	Score	Error	Units
DepthStackCallPerformanceCheck.benchException	0	avgt	25	1060,989	± 11,319	ns/op
DepthStackCallPerformanceCheck.benchException	5	avgt	25	1494,899	± 22,348	ns/op
DepthStackCallPerformanceCheck.benchException	10	avgt	25	1980,053	± 15,842	ns/op
DepthStackCallPerformanceCheck.benchException	20	avgt	25	3111,082	± 52,430	ns/op
DepthStackCallPerformanceCheck.benchException	50	avgt	25	5869,144	± 76,004	ns/op
DepthStackCallPerformanceCheck.benchException	100	avgt	25	10887,232	± 139,166	ns/op
DepthStackCallPerformanceCheck.benchException	150	avgt	25	15664,843	± 211,139	ns/op
DepthStackCallPerformanceCheck.benchNotEx	0	avgt	25	3,940	± 0,016	ns/op
DepthStackCallPerformanceCheck.benchNotEx	5	avgt	25	10,598	± 0,068	ns/op
DepthStackCallPerformanceCheck.benchNotEx	10	avgt	25	13,373	± 0,187	ns/op
DepthStackCallPerformanceCheck.benchNotEx	20	avgt	25	18,269	± 0,153	ns/op
DepthStackCallPerformanceCheck.benchNotEx	50	avgt	25	37,355	± 0,297	ns/op
DepthStackCallPerformanceCheck.benchNotEx	100	avgt	25	74,598	± 0,327	ns/op
DepthStackCallPerformanceCheck.benchNotEx	150	avgt	25	106,736	± 0,578	ns/op

```
private Integer withException(int depth) {
    if (depth == 0)
        throw new ArithmeticException();
    else
        return withException(depth - 1);
}

private Result<Integer, String> withNotEx(int
 ↪  depth) {
    if (depth == 0)
        return
         ↪  Result.err(Notification.of("Arithmetic
         ↪  error"));
    else
        return withNotEx(depth - 1);
}
```

Listing 7: Methods used to compare nested calls

6. Validation and JSR 303

NotEx is applicable where an exception is not required. One of the special cases is validation, which NotEx copes with perfectly.

Unlike exceptions, NotEx allows you to process all fields, not just one, which gives a significant advantage. Listing 8 shows an example of using errorObject, which has the onEachError method that allows the user to go through all the errors

```
UserFields userFields
      =new UserFields(1, -1, 1, -1);
boolean valid = validate(userFields)
 .resolve(errorObject -> {
  errorObject.onEachError((field, error) ->
   printf("Error on '%s' : %s%n", field,
    ↪  error));
   return false;
});
```

Listing 8: Simple validation example

In Java language since 2009, there is a JSR 303: Bean Validation [8] specification, which defines a meta-data model and API for JavaBean validation based on annotations

NotEx has no special limitations, since for the most part it acts as a wrapper that allows you to handle errors linearly. Therefore, integration with JSR 303 is available, which will allow validating fields through annotations. Example is given in Listing 10 and 9

```
private static class User {
    @Min(10)
    private final int age;
    @NotNull
    private final String name;

    public User(int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

Listing 9: Validation class definition

```
User user = new User(5, null);

boolean isValid =
 ↪  NotExValidator.validate(user)
.resolve(validationError -> {
 validationError.onEachError((field, msg) ->
  printf("Error on field %s : %s%n", field,
   ↪  msg));
  return false;
});
```

Listing 10: Processing

7. Linear Execution

An exception has the property of non-linearity when it is handled elsewhere depending on the order of method invocation when an exception is thrown

NotEx allows linear code in which all errors are handled in the tail section, which reduces the complexity of the code

```
Integer number = 0;

String result = Result.<Integer,
 ↪  String>of(number)
 .apply(this::minusOne)
 .apply(this::minusTwo)
 .apply(this::minusThree)
 .resolve(num -> "Number is " + num,
 errorObject -> errorObject);
```

Listing 11: Processing

8. Conclusion and Future work

As it has been shown, the standard exception mechanism in Java programming language is not very well suited for some scenarios. Based on the notification pattern a different approach for passing extended information about errors was proposed, which can be used for handling expected exceptional conditions. This approach uses the Result object and can contain multiple notifications about an unsuccessful operation on user input, external data input and other similar information objects.

Using the proposed approach the NotEx library was created. NotEx can greatly improve performance and make the code cleaner, more linear and can be easily integrated with some Java extensions such as Bean Validation and Stream API.

Some experiments were delivered to compare the performance of the developed library with the standard Java exception handling. The experiment results show that the NotEx library has a performance boost up to more than 250 times compared with traditional exception handling, especially when nested calls are used. Moreover, the library can pass multiple notifications increasing the amount of information passed to the calling module.

In future works, it is planned to improve the functionality of the library by adding the ability to pass the custom error handlers to the called function in order to move all handling code to the place where errors are generated, and the ability to stack notifications like the standard exception implementation.

References

[1] G. Petrosyan, Language support for systematic error handling, 2013. URL: https://infocom.spbstu.ru/article/2013.34.7/.

[2] A. F. Garcia, C. M. Rubira, A. Romanovsky, J. Xu, A comparative study of exception handling mechanisms for building dependable object-oriented software, Journal of Systems and Software 59 (2001) 197–222. URL: https://www.sciencedirect.com/science/article/pii/S0164121201000620. doi:https://doi.org/10.1016/S0164-1212(01)00062-0.

[3] J. B. Goodenough, Exception handling: Issues and a proposed notation, Commun. ACM 18 (1975) 683–696. URL: https://doi.org/10.1145/361227.361230. doi:10.1145/361227.361230.

[4] J.-W. Jo, B.-m. Chang, K. Yi, K.-m. Choe, An uncaught exception analysis for java, Journal of Systems and Software 72 (2002). doi:10.1016/S0164-1212(03)00057-8.

[5] F. Ebert, F. Castor, A. Serebrenik, An exploratory study on exception handling bugs in java programs, Journal of Systems and Software 106 (2015) 82–101. URL: https://www.sciencedirect.com/science/article/pii/S0164121215000862. doi:https://doi.org/10.1016/j.jss.2015.04.066.

[6] A. Volushkova, Analysis of exception handling in different programming languages, 2014. URL: https://vestnik-muiv.ru/en/article/analiz-raboty-s-isklyucheniyami-v-razlichnykh-yazykakh-programmirovani

[7] M. Fowler, Replacing throwing exceptions with notification in validations, 2014. URL: https://martinfowler.com/articles/replaceThrowWithNotification.html.

[8] JCP, Jsr 303, 2009. URL: https://jcp.org/en/jsr/detail?id=303.