Improvement of Software Diagnostics with Advanced Execution Logging

Andrei-Ionuţ Nicolaev
Transylvania Bank
30-36 Dorobantilor Street
andrei.nicolaev@btrl.ro

Crenguţa-Mădălina Puchianu
Ovidius University
124 Mamaia Blvd.
crenguta.puchianu@365.univ-ovidius.ro

ABSTRACT
In this paper we present the development process of the Execution Lens, a software system that solves the problems of monitoring and re-running the processes of another application, with the aim of improving its quality and enhancing bug solving efficiency. The system consists of two packages intended for applications developed using C#/.NET and a web application for viewing and managing data saved for monitoring. Packages are in use to save the data needed to be monitored and respectively to be able to re-execute flows of the host application. The system was tested from the point of view of usability by a group of developers with at least one year of experience in the field. The result of the evaluation was a good one, the stakeholders being satisfied with the Execution Lens, finding it useful in the development of their applications.

Author Keywords
software engineering, software logging
ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

DOI: 10.37789/ icusi.2024.27

INTRODUCTION

Due to the increase in the complexity of software systems, the need to monitor their execution has also increased. For this reason, log files or execution files appeared, with the aim of improving their performance and efficiency.
There are studies ([1] and [2]) that show that code logging is still in development, a fact reinforced by the lack of a standard, respectively a guide for logging practices accepted by the software developer community. Thus, each developer must rely on personal experience or intuition to decide what information is important and where it should be saved. For example, many developers do not know how to log information in an application and choose to do this directly in the source code, which is often part of the business logic of the application. This practice must be avoided, because it can lead to inefficient logging and more difficult debugging of the application, and instead an independent subsystem specialized in information logging should be created.

There are cases in software applications where a bug is hard or even imposible to reproduce, this event usually happens when logs provide incomplete information or the system changed its state. Evenmore in domains like banking where there are applied penalities on how much time the system went down or had a semnificative problem, finding the root cause of the problem and coming with a solution should be done fast.

This paper proposes the Execution Lens, a software system for monitoring and re-executing the processes of another application, which solves those problems by storing the internal state of the application and re-executing the process when needed.

The remainder of the paper is organized as follows. The following section presents some of the models and software components made during the development of the system. This section is followed by Related work that contains a comparison of the system with two other representative applications in the field and similar to Execution Lens from the point of view of the implemented functionalities. Then, Execution Lens was tested in terms of performance and validated by a group of developers, and the results obtained are presented in the Testing and Validation section. The paper ends with a section of conclusions and future research.

Development process

Software Analysis

A software actor is a role "played" by one or more individuals (person, team, or organization) or even another software application in its interactions with our application. The role is is characterized by a set of properties and actions which each individual in this role can exhibit or "plays" in the given context [3]. In the case of the Execution Lens application, we identified one software actor: developer.

Software Use Case Diagram

The software use case diagram belongs to the functional model created during the object-oriented analysis activity of a software system. Besides the software use case diagram, a functional model contains the description of the software use cases and their activity diagrams [3].
The software use case diagram is formed by the software actors and uses cases and their relations. The Execution Lens system has the software use case diagram presented in Figure 1.

Software Design and Implementation

The object-oriented analysis has focused on learning to "do the right thing"; that is, the understanding of the goals for the Execution Lens application, and related rules and constraints. By contrast, the design work will stress "do the thing right" [3]; that is, skillfully designing a solution to satisfy the system goals.

The heart of this solution is the creation of the system software architecture, which contains on high level the design component and their relations.
Execution Lens contains five software components (Figure 2) which will be briefly explained below.

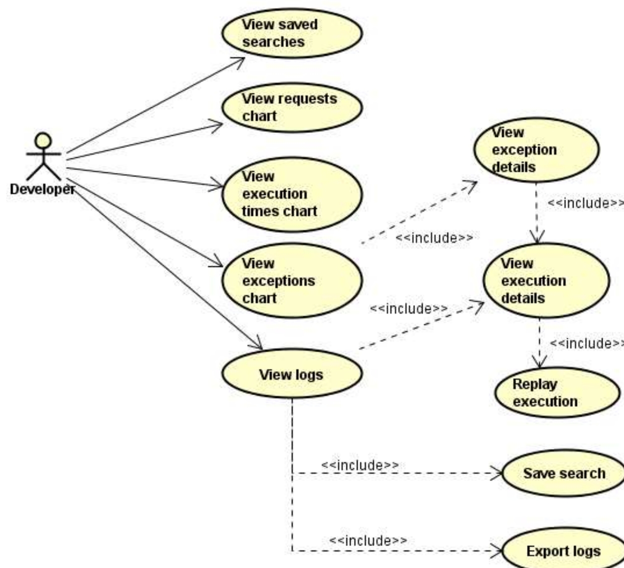

Figure 1. The Unified Modeling Language (UML) [4] software use case diagram of the Execution Lens system

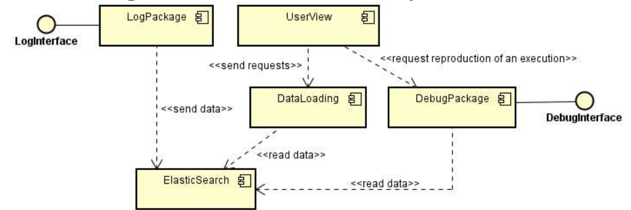

Figure 2. The UML component diagram of the Execution Lens system

Our system provides the LogInterface interface through which the host application under development or in production, must connect. To achieve dynamic and non-intrusive information logging, LogPackage applies the Dynamic Proxy [5] design model to create a proxy object of the class to be logged. In this way, function calls can be intercepted along with their data and the information can be logged.

To structure the information from methods, we used a tree-type structure to keep the order of the method calls. However, as between the input parameters of the methods and their results there may be other methods calls, we used a stack for the temporary storage of the methods. Thus, when the result provided by the method is received, the current method will be removed from the stack and its result will be added to the tree.

When the program reaches back to the execution start method and there will be no more calls or instructions to execute, the tree will be complete, and each of its nodes will have a reference to the parent node. The nodes will then be indexed in Elastic Search [6] using a recursive method. Since we chose to use Elastic Search for data storage, the creation of the index was not necessary, it is being created automatically at the first call of the indexing method. Within the indexing, we chose to use a single index to keep the data, and for each node in the tree we created a new document. At the end of the indexing, the ID of the root node will be returned, which will be added to the HTTP request to be used later in case of an error.

In the object-level design of DebugPackage, we also used Dynamic Proxy [5] to create a consistency interceptor of the inputs and outputs of the methods and then of the proxy objects. To reproduce an execution, it is necessary to have the ID of a node, whether it is the root or not, after which a recursive method will be called to retrieve the data and rebuild the tree.

Once we have the complete tree, a recursive method is called, which based on the input parameters, the class, the methods, the provided result, will dynamically create the object and create step by step from the leaf to the root all the dependencies necessary for re-execution.

The consistency interceptor is a very important component, because regardless of the values received by the methods or their results, it will modify them with the values from the time of the original execution. In this way, it is ensured that debugging will be done correctly and the values will be the correct ones. An important aspect to specify is the fact that since the data is stored in Elastic Search, in the form of JSON documents, when the data is retrieved and deserialized, the object type will be lost. To solve this problem, we took the types of parameters and results from the method and converted the values from JSON into the respective types. When the root instance is created, the execution start method will be called and execution debugging will begin.
The Debug Package provides the Debug Interface through which the tested host application is linked to the Execution Lens.

Furthermore, the developer can use the front-end of the Execution Lens system to have access to other functionalities of the system such as the search function, viewing the data saved after performing some searches, viewing the execution times of the methods, viewing the sequence diagram of a method execution and viewing the methods that threw exceptions (Figure 3).
The Execution Lens system offers three ways to search for data, as can be seen in Figure 4. These will be briefly presented below.

a) the search in natural language does not require compliance with a certain format and performs the search based on the words written by the user. If no results are obtained, a message is displayed with the reason for the failure.

b) search by simple filters such as: the time period in which the desired classes or methods were tested or the logs that appeared. Also, complex filters can be created using the operators presented in Figure 5. These operators are applied to the input and output parameters of the methods and to the searched logs.

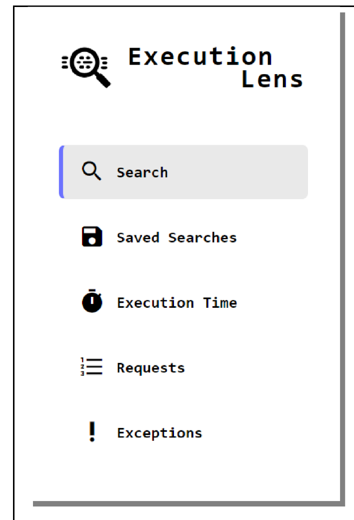

Figure 3. The main menu of the Execution Lens system

c) search by the ID of a log, in case the developer wants to correlate an action or an event with the ID of a log. For example, in the case of a bank transaction, the respective transaction can have the ID of a log correlated and thus all the details during the processing of the transaction can be seen.

Search results can be saved in the database or exported to files.

Another important aspect is that the system displays dynamic sequence diagrams of objects that interact with each other in different scenarios of the tested use case. For example, Figure 6 shows the sequence diagram of the objects of the BookReservation use case, which stops at the ValidateReservation method because it threw an exception. In this case, the output of this method will be marked in red and the thrown exception will also be displayed. More information is displayed if the user clicks on the marked replay messages (Figure 7).

The sequence diagrams were implemented using the mermaid.js [7] and panzoom.js [8] library packages. The second library was used to allow the user to move the diagram from one part to another and to enlarge a part of the diagram.

Another functionality of Execution Lens is the possibility of re-creating the execution, in order to be

able to troubleshoot any problems that may arise in the interaction of users with the host application. When the Replay button is pressed, the execution will be rebuilt based on the information stored in the database, after which this execution will be executed, allowing the developer to debug the code without having to prepare the environment for reproducing the error, which often takes time. It's up to each developer which IDE he/she wants to use for debugging, this not being a constraint imposed by the system. This feature is an important one because we get an insight of what happened on the production environment on a specific execution use case.

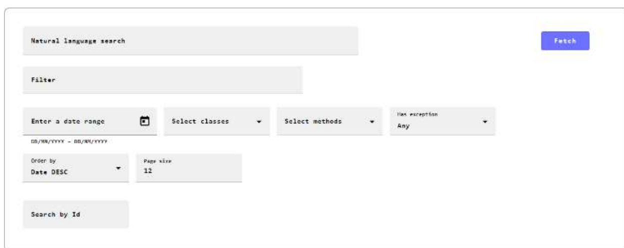

Figure 4. Ways of performing the data search

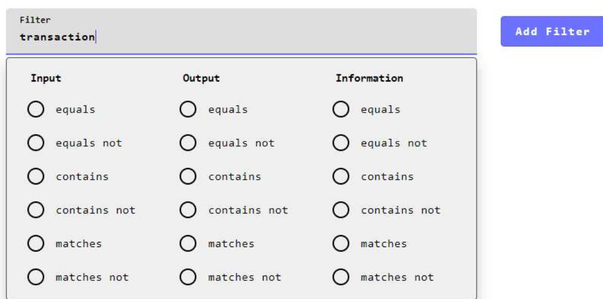

Figure 5. Graphical interface for creating search filters

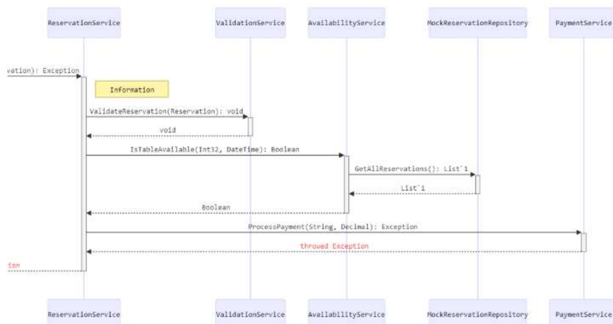

Figure 6. Sequence diagram of a scenario in which an exception occurs

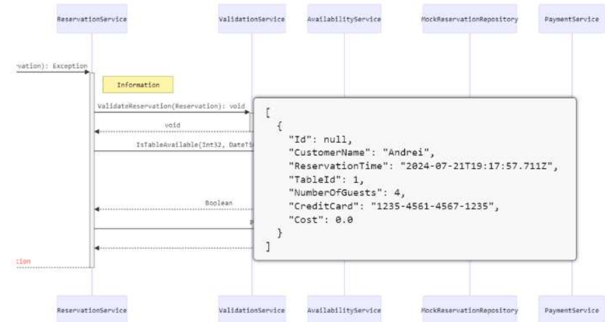

Figure 7. Displaying information about an exception that occurred in the execution of a scenario

On the backend part of the system for retrieving the data to be displayed in the graphs, we used Elastic Search with NEST, a C# client for Elastic Search. We used filters, aggregation scripts and queries to process and retrieve the data needed for display. For instance, Figure 8 shows in pie form the execution times of each method called during the execution of the BookReservation method.

Testing and validation

Performance testing

In order to test the performance of Execution Lens, we used BenchmarkDotNet[9] which is a .NET library for benchmarking with many features.

With this tool, we designed a benchmark that measured (in microseconds) the average execution time, the error and the standard deviation for scenarios where Execution Lens does not save and saves logged methods in the database. Having the error, we calculated the confidence interval according to the formula [mean-error, mean+error]. We ran BenchmarkDotNet to provide these values for a number of 10, 100 and 1000 raw and logged methods. The raw methods are the basic application methods that were intercepted by our application.

The results obtained are presented in Table 1. From the analysis of these values, we can conclude the following:
- when Execution Lens does not save logged methods in the database, the application has approximately the same average execution time as the base application.
- when Execution Lens uses the database, the performance of the application decreases compared to the base application, as the number of methods run increases. Thus, for 100 tested methods, Execution Lens executes the logged methods 2 times slower than

the base application, the ratio being 0.58. Moreover, for 1000 methods tested, Execution Lens executes the logged methods 8 times slower than the base application, the ratio being 0.12.

Usability evaluation

Measuring the quality attribute of usability is a key factor for any software product [10], particularly for Execution Lens.
The evaluation of the system was done on a group of 23 developers which evaluated the system. The respondents have mainly an experience between 1-3 years, but there were also stakeholders with an experience of less than one year and more than 7 years of work in programming .NET applications.
They completed a questionnaire consisting of 10 questions, which we list in Table 2.
From the analysis of the respondents' answers, we can conclude that the system has good usability aspects, which will allow developers to carry out debugging and monitoring activities faster and more efficiently.

RELATED WORK

At this moment, there is a limited number of applications and libraries on the market to satisfy the requirements of all developers. For this reason, this gap represents an opportunity for new applications and libraries to come up with better and more efficient solutions.

In this subsection, we will present some of the rival applications and libraries that promise to fulfill the basic requirements in this field.

Serilog

Serilog is a popular framework for structured logging in .NET, with over 1.2 billion downloads [11]. It has gained popularity because it is easy to install and does not require additional configurations to start saving information, but it also comes with a wide range of extensions and configurations such as enriching the logs with information, and such as the machine on running application, application environment, threads, client data and more.

An advantage that the framework offers is that it offers the possibility to log the information in batches to reduce the number of operations and to increase the performance of the application.

UndoDB [12] is a code execution monitoring application that provides the ability to monitor and revert to the state of the application during runtime. This application is used by developers to find and fix bugs that cannot be consistently reproduced or to better understand how the application works at runtime.

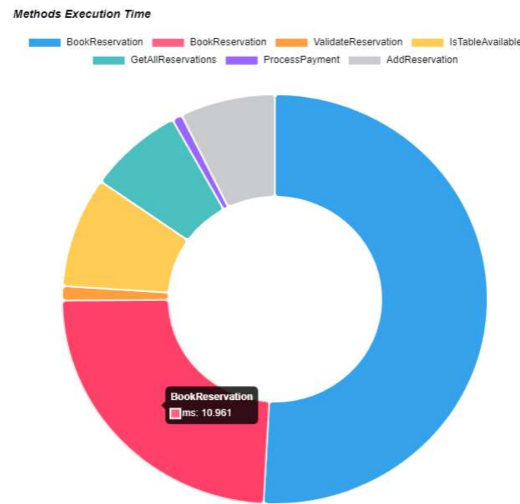

Figure 8. Execution time graph of the BookReservation method

Utilizability questionnaire
What functionalities of the system do you prefer?
Does the system have all the expected functionalities?
How do you rate the overall UI of the system?
How much do you think this system makes your work as a developer more efficient?
Have you encountered any execution problems (errors, blockages or delays) of the application?
How much would you use the application in the projects you are working on?
Do you consider that the application is intuitive and easy to use?
Would you recommend the application to other developers?
Please write 3 things you liked about this system.
Please write 3 things you did not like about this system.

Table 2. The questionnaire used in the usability testing of the Execution Lens system

UndoDB

The notable benefit that the application brings is the reduction of the time needed to prepare the system for reproducing the bug and for finding it. UndoDB allows memory inspection, conditional breakpoints, monitoring points, variable inspection, all of which are available during runtime, both continuing execution and returning to a previous point. Execution debugging is available in the application's own interface, as well as in IDEs such as Visual Studio or

Eclipse by installing the UndoDB extension.

Comparativ table

Each of the applications presented above fulfills certain functions also present in Execution Lens, but none of them offers all the desired functionalities. The Table 3 shows the key features of Execution Lens and of the two applications described above.

Functionality	Execution Lens	Serilog	UndoDB
Information logging	√	√	√
Reproduction of executions	√	X	√
Data visualization	√	X	X
Detailed view of exceptions	√	X	X
Viewing the execution times of the functions	√	X	X
Detailed view of the execution	√	X	X
Notifications in specified cases	√	√	X
Correlation of an execution with a log	√	√	X
Writing logs in batches	√	√	√

Table 3. Comparison of the Execution Lens application with two other similar applications

Conclusion and future work

Based on the conducted experiments on voluntary subjects the Execution Lens system presented in this paper proved that it is a useful tool for developers in order to make the debugging and monitoring process of the applications that they develop more efficient.

We will continue to enhance its functionality by adding other requirements, such as: obfuscation of personal data, implementation of packages for asynchronous applications, training of automatic learning models for determining abnormal executions, and migration to several types of applications developed in C#/.NET.

REFERENCES

10. Pecchia, A., Cinque, M., Carrozza, G., and Cotroneo, D. Industry practices and event logging: Assessment of a critical software development process. In 2015 *IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 169–178. IEEE, (2015).
11. Fu, Q., Zhu, J., Hu, W., Lou, J-G, Ding, R, Lin, Q., Zhang, D., and Xie, T. Where do developers log? An empirical study on logging practices in industry. *Proceedings of the 36th International Conference on Software Engineering*, 24–33. ACM, (2014).
12. Pressman, R. S., Software Engineering. A Practitioner's Approach, *McGraw-Hill Publishing Company*, (2000).
13. OMG, Unified Modelling Language, version 2.5.1, https://www.omg.org/spec/UML/, (2017).
14. Freeman, E., Hupfer, S. and Arnold, K. JavaSpaces Principles, Patterns, and Practice. *Addison-Wesley Professional*, (2004).
15. Elastic.co. Elastic search, https://www.elastic.co/guide/en/elasticsearch/reference/7.17/index.html.
16. Mermaid diagramming and charting tool, https://mermaid.js.org/
17. Panning-zooming-any-elements, https://www.jqueryscript.net/zoom/jQuery-Plugin-For-Panning-Zooming-Any-Elements-panzoom.html
18. .NET foundation, BenchmarkDotNet, https://benchmarkdotnet.org/
19. Abran, A., Khelifi, A., Witold, S., Seffah, A. Usability Meanings and Interpretations in ISO Standards, *Journal of Software Quality*, 11(4), 325-338, (2003).
20. Serilog, http://nuget.org/packages/Serilog/4.0.1-dev-02205
21. UndoDB, Undo, https://undo.io/resources/undodb-reversible-debugging-tool-linux-and-android

Number of intercepted methods	The kind of method	Without persistence	99% confidence interval		StdDev	With persistence	99% confidence interval		StdDev
10	raw	995.8 ms	983.43 ms	1008.17 ms	10.96 ms	972.1 ms	962.62 ms	981.58 ms	7.91 ms
	logged	994.7 ms	977.88 ms	1011.52 ms	15.74 ms	1067.8 ms	1051.1 ms	1084.5 ms	14.8 ms
	ratio	**1.001105861**				**0.910376475**			
100	raw	1008.6 ms	993.53 ms	1023.67 ms	13.36 ms	958.1 ms	949.19 ms	967.01 ms	7.9 ms
	logged	991.8 ms	973.04 ms	1010.56 ms	19.27 ms	1640.3 ms	1669.26 ms	1669.26 ms	43.35 ms
	ratio	**1.016938899**				**0.584100469**			
1000	raw	1014.6 ms	998.9 ms	1030.3 ms	14.68 ms	983.3 ms	969.72 ms	996.88 ms	11.34 ms
	logged	1111.1 ms	1098.58 ms	1123.62 ms	11.1 ms	7735.8 ms	7625.6 ms	7846 ms	86.04 ms
	ratio	**0.913149131**				**0.127110318**			

Table 1. The results of testing the performance of the Execution Lens system in relation to the basic application