MOD002602 Computing Research Methodologies

Element 010: 3000 WORD REPORT (2021 TRI2 F01CAM)

PROJECT PROPOSAL TEMPLATE

Prepared by

Student Name and ID: Andrew Christian Ackerman 2103400

Project Title

Debugging an Aspect Oriented Program.

Submission Date

28/04/2023

Author name: Andrew Christian Ackerman

Table of Contents

Abstract

Aspect Oriented Programming (AOP) aims to solve and implement non-function requirements quickly. AOP programs are hard to debug due to their complexity. This paper aims to reduce the difficulty of debugging to lower the barrier of entry to coding with AOP. This paper creates a prototype debugger that achieves fault isolation and then evaluates it experimentally by looking at its accuracy. We then compare it to how fast it works compared to the best current debugger and no debugger. We find that debugging a static Aspect-oriented program is an underresearched subject, and more work needs to be done on fault isolation. This research will help develop debugging tools to make it easier to debug an AOP program, making it easier to learn and thus saving time for researchers and businesses.

Part A: Project Context (Deadline : Week 6 Teaching week)

The purpose of Part A is to provide an overview of the project context and business context.

1. Introduction

Non-function requirements, like debuggability, security, logging, code readability, and efficiency, are essential in computing. However, implementing them requires code reuse across many modules and methods, which requires maintenance, and may sacrifice efficiency or readability(Kiczales et al., 1997). Aspect Oriented Programming (AOP) is a field of study that addresses those non-function requirements without losing efficiency and maintainability. It creates aspects (the non-function requirements) and combines those aspects with the original code, with the output meeting the non-functional requirements.

More specifically, AOP searches for some condition(called the point cut) to insert the action to be executed (the advice, think of it as the aspect code) into the base program. The place where the code will be inserted is called the join point (baeldung, 2023). Pointcuts could be code structure or a change in the control flow (e.g., if, else, change the control flow). (Haupt et al., 2005)

As the code becomes more complex, mistakes will occur, and it will be more challenging to debug the code, which is complicated by adding AOP, as the bug may not be in the source code, but in the combined code with the aspects. This paper aims to reduce the debugging challenge and the barrier to coding entry with AOP.

2. Research Questions

What does it mean to debug an Aspect-Oriented program?

What processes would aid the process of locating a bug?

What features of Aspect-Oriented languages would be subject to debugging?

Are there any features in the programming language that would reduce the amount or complexity of debugging an AOP program?

What information would be helpful in aid programmers in debugging an AOP program? Are those the same as error messages?

What are the current solutions to debugging an AOP program? What progress has already been made?

3. Research Aim

I want to make it easier to debug Aspect-Oriented programs to decrease the barrier to entry for metaprogramming. This will, in turn, increase the popularity of AOP in the field.

4. Research Objectives/ Deliverables

Find out unsolved problems in debugging an AOP.

Define what areas the prototype debugger will and will not cover.

Learn how to code in an aspect-oriented language, as student 2103400 should have some familiarity with the subject that needs to be debugged.

Create a prototype debugger for an Aspect Oriented program.

Evaluate the debugger using real code examples.

5. Business Need/Case

- **Problem background**

  Debugging a program is costly (Planning, 2002), and having proper debugging software can speed up the development time of a program, hence reducing the cost of experienced programmers. Debugging tools makes it easier to learn AOP, which decrease the initial cost of learning how to program an Aspect-Oriented program. However current debugging infrastructure is insufficient for AOP needs.

- **Benefits**

  It could reduce costs, speed up development time, make it easier to learn programs, which would increase the popularity of the program itself.

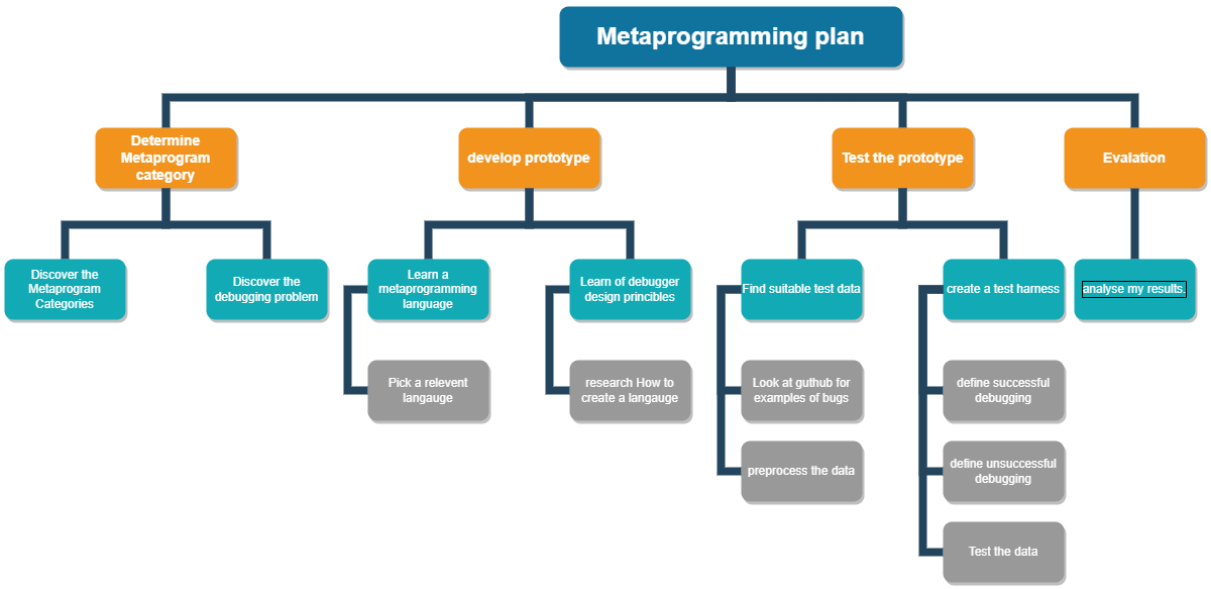
- **Dis-Benefits**

As this is a student project, the scope of the project will be limited, to ensure the completion of the project. This means that my software will only solve the a very specific issue, at the cost of everything else. This means the software may not efficient, or scalable, reducing its use in a business environment. It may have compatibility issues with other debugging software that offer things that the prototype doesn't, meaning developers might have to choose between 2 flawed choices.

- **Approximate Budget / Time**

6. Gantt Chart

7. Work Breakdown Structure(WBS)


Metaprogramming plan
Determine Metaprogram category
develop prototype
Test the prototype
Evalation
Discover the Metaprogram Categories
Discover the debugging problem
Learn a metaprogramming language
Learn of debugger design princibles
Find suitable test data
create a test harness
analyse my results
Pick a relevent langauge
research How to create a langauge
Look at guthub for examples of bugs
define successful debugging
preprocess the data
define unsuccessful debugging
Test the data

End of Part A

Part B: Review of Literature and Research Methodology

(Deadline : according to course work submission deadline)

The purpose of part B is to review the literature review and underlying methodology planning to use by this research.

8. Literature Review

Keywords: debugging, metaprogramming, Aspect oriented programming, multistage programming

It is a descriptive, theoretical paper. (Eaddy et al., 2007) creates a debugging model and a set of properties that an excellent debugging model should have. The Assumptions appear to be correct and well-referenced when they are not intuitive. It evaluates current debugging software on how well it satisfies those properties and tests its software against an example. However, the model appears to be focused on dynamic AOP instead of static AOP, as shown by P4, P5, and P6. P4, being able to turn on and off aspects at runtime to help identify problematic AOP by omission; P5, being able to add new Aspects "in an unanticipated fashion," e.g., at runtime; and P6, being able to change the code of the aspects or the non-meta while the computer is running. P4, P5, and P6 are all runtime-focused. It also found that no current debugging model implemented fault isolation, as shown in Figure 2 in that paper. Also, one problem with fault isolation is that some AOP languages break library boundary ownership, i.e., if an error is found in a library, a class, etc., the cause of the error is inside that library, or class, etc., as AOP combines code.

(Lilis and Savidis, 2020) Gives a review of the current state of metaprogramming (which is a program that takes another program as input) and classifies the respective fields in a way consistent with already used terminologies. It finally evaluates what languages support metaprogramming capabilities. There is an equal distribution of languages that support dynamic or static AOP systems. Lilis and Savidis also found that static AOP occurs mainly in compile or pre-processing time, while Dynamic AOP mainly occurs in runtime. This shows a need for Compile time Aspect-oriented programs that are not covered by (Eaddy et al., 2007).

There is a lack of research based on empirically evaluating debugging models.(Parizi et al., 2011) Evaluates current automated testing approaches based on "*fault detection effectiveness*." It spent a lot of effort designing an experiment to test the current automatic AOP results.

(Yin, 2013) is mainly about visualizing the runtime information that occurred graphically. They represent a way to query the program's behavior graphically, allowing the programmer to ask the program which advice matches what bit of code. This enables the programmer to investigate the program's behavior and see if the advice only applies to where it is meant to be used. However, it may violate the property of debug Obliviousness (Eaddy et al., 2007), though that is by design. This is useful for omniscient debugging.

(van't Riet, Yin and Bockisch, 2013) Omniscient debugging is when the program captures all interesting information at runtime, so the programmer can look back on that for debugging. They define what events will be recorded using "*advanced-dispatching language*" concepts. They created a way to quickly navigate through all the gathered runtime

information using a graphical interface in different levels of detail. The way they summarized the information, to be able to question the program's execution history was well thought out. It did target its designed UI to two AOP-specific scenarios. However, this appears to be targeted at dynamic AOP, not static AOP, as it gathers runtime information. Also, it doesn't cover what happens if an error occurs. The erroneous code would likely reflect a change in the execution history shown in the GUI, but that requires the programmer to find the mistake 1st, notice it exists, and then track it down. It doesn't give any hints when an error has occurred or where it would be, without knowing what normal execution flow would be. In other words, it fails at fault isolation. Also, they haven't implemented the UI yet, as they created a design, so there are no empirical tests on whether the designed system will save time yet.

(Jezequel, 2010) does add execution levels to AOP. Aspects at a higher level can insert code into aspects of a low level, but not vice versa. This fixed some bugs, like an aspect trying to edit its code infinitely. However, the conclusion linked this to runtime programming. Unsure if it could be translated to compile-time metaprograms.

(Lilis and Savidis, 2013) Focused on creating a debugging interface for a multistage program, another metaprogramming category. They found that aspects they took advantage of when designing the debugging software were "*not tightly coupled with metaprogramming*," and they would investigate how it affects other metaprogramming fields like AOP. However, the interface looks promising, as it allows the programmer to track a chain of errors as the program transforms itself automatically, allowing the programmer to track the error down easily. It also focused on compile-time metaprogramming, though they had thoughts on extending it to runtime metaprogramming. They boldly assumed that the error message itself did not need modification.

9. Research Methodology

9.1 Research Framework

I will need to decide what different categories of code there are for fault isolation, e.g., "*base code, advice code or other AOP activity code*" (Eaddy et al., 2007)

9.2 Data Collection

After the prototype has been developed, two things need to be tested.

1. Do the prototype's fault isolation and other functions work as intended, and how accurate is it? First, collect example AOP code with errors, label where the error was, and label what category of code the error was caused from(advice code, base code etc.). Ensure the example AOP code reflects different scenarios to ensure the experiment's validity. For example, if 5 code examples are essentially the same but have slightly different syntax, they will test the same thing and won't test if the debugger works with a range of different code. Then feed the prototype debugger the example code. Record what the prototype debugger thinks of where the error is. Then plot a confusion matrix, and record the accuracy, precision r2 score, etc.

2. Does the debugger save time in the real world? 1st, create a set of problems to be solved with AOP code. Find a group of programmers to experiment on(need more work on this). Split the group into three categories. One group will be told to do the activities without any debugger. The 2nd group will be told to do the activities with the prototype debugger. The 3rd group will be told to do the activities with the best AOP debugger that is currently available to use. The time that each person does each activity will be recorded. Then compare the results of each group.

9.3 Results and Discussion

I expect that in experiment 2, the prototype debugger should perform better than no debugger but worse than the best debugging software for AOP.

There is a lack of research on debugging Aspect-oriented programs. The papers that were reviewed focused more on runtime AOP, and hence dynamic AOP(Lilis and Savidis, 2020) rather than compile time AOP and thus static AOP.

There were many ways to make debugging easier. (Jezequel, 2010) designed the language to eliminate common errors elegantly, compared to other languages, making it more understandable, and means there is less information to keep track of, reducing the bugs that occur. (Kiczales et al., 1997) Designed an architecture that clarifies what properties a debugging system should have and common errors to care about. (Yin, 2013) created a system that allows a programmer to question how the aspects pick where to insert themselves, allowing the programmer to question the program and increasing its comprehensibility. (van't Riet, Yin and Bockisch, 2013) recorded all interesting runtime information, allowing a programmer to see the program history. This means they can see if the program applied the aspects correctly by looking at the program's state, and if an error occurs, it allows a programmer to see how the aspects were incorrectly applied to the program. While (Parizi et al., 2011) focused more on automating the AOP testing process, which could decrease costs.

This project has found that there was a lot of effort placed on being able to ask the program questions (Eaddy et al., 2007; van't Riet, Yin and Bockisch, 2013; Yin, 2013), there was not as much effort on fault isolation, and if an error occurs, giving a hint about where the error occurred. This is shown by Table 2 from (Eaddy et al., 2007), which shows that no current program debugger implements fault isolation, including their own. P4, P5, and P6 from (Eaddy et al., 2007) help a programmer deduce where the error occurred, as they aim to give the programmer an insight into how the code was executed, but the programmer still needs to find the error. Same problems for (van't Riet, Yin and Bockisch, 2013; Yin, 2013). (Lilis and Savidis, 2013) does give the programmer a direct way to track an error chain directly and automatically. However, they mainly focused on a multistage program rather than an Aspect Oriented program, so more work still needs to be done on this subject.

Table 1 by (Eaddy et al., 2007)

AOP Tools & Systems	Idempotence	Debug intimacy	Debug obliviousness	Dynamism	Aspect introduction	Runtime modification	Fault isolat./repud.	
AOP.NET	✓	O			✓			
AOP-Engine		O			✓	✓		
Arachne	✓				✓	✓		
AspectJ	✓	O						
AspectWerkz	✓	O	¤		✓			
Axon	✓	O	¤		✓			
CaesarJ	✓							
CAMEO	✓	✓						
CLAW		O			✓			
EAOP	✓	O	¤		✓	✓		
Handi-Wrap	✓				✓	✓		
Hyper/J	✓	✓						
JAsCo		O	¤		✓	✓		
nitrO		O			✓	✓		
Wicca v1	✓	✓			✓	✓	✓	
PROSE v2	✓	O	¤		✓	✓		
SourceWeave.NET	✓	✓						
Steamloom	✓	O	¤		✓	✓		
Wool	✓	O	¤		✓	✓		

✓- Fully supported
O- Partial *advice execution* debugging supported
¤- Partial obliviousness supported

Table 2. AOP debuggability comparison matrix. Our system, Wicca, is shown in bold

10. Conclusion

We found that debugging AOP programs is lacking. We found a model that answers what is means to debug an AOP language, what AOP activities will be subject to the debugging(Eaddy et al., 2007). We found language features that aid the debugging process(Jezequel, 2010). We found many different ways of debugging an AOP program. We found that that research on debugging static AOP is serious lacking, and the fault isolation is an area that needs more research. This project tried to make a prototype debugger and evaluating the debuggers validity experimentally. Future work would be focuses on making the prototype debugger compatible with other debuggers, and making the debugger more computationally efficient, and easier to use, and creating a debugging model for compile time Aspect Oriented programming.

References

baeldung, 2023. *Introduction to Spring AOP | Baeldung*. [online] Available at: <https://www.baeldung.com/spring-aop> [Accessed 20 March 2023].

Eaddy, M., Aho, A., Hu, W., McDonald, P. and Burger, J., 2007. *Debugging aspect-enabled programs*. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, https://doi.org/10.1007/978-3-540-77351-1_17.

Haupt, M., Mezini, M., Bockisch, C., Dinkelaker, T., Eichberg, M. and Krebs, M., 2005. An execution layer for aspect-oriented programming languages. In: *Proceedings of the First ACM/USENIX International Conference on Virual Execution Environments, VEE 05*. pp.142–152. https://doi.org/10.1145/1064979.1065000.

Jezequel, J.-M., 2010. Execution levels for aspect-oriented programming. In: *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10: Ninth International Conference on Aspect-Oriented Software Development. [Place of publication not identified]: Association for Computing Machinery. pp.620–48. https://doi.org/10.1145/1739230.1739236.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J., 1997. *Aspect-oriented programming*. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, https://doi.org/10.1007/bfb0053381.

Lilis, Y. and Savidis, A., 2013. An integrated approach to source level debugging and compile error reporting in metaprograms. *Journal of Object Technology*, [online] 12(3). https://doi.org/10.5381/jot.2013.12.3.a1.

Lilis, Y. and Savidis, A., 2020. A Survey of Metaprogramming Languages. *ACM computing surveys*, 52(6), pp.1–39. https://doi.org/10.1145/3354584.

Parizi, R.M., Ghani, A.A.A., Abdullah, R. and Atan, R., 2011. Empirical evaluation of the fault detection effectiveness and test effort efficiency of the automated AOP testing approaches. *Information and Software Technology*, 53(10), pp.1062–1083. https://doi.org/10.1016/J.INFSOF.2011.05.004.

Planning, S., 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 1.

van't Riet, M., Yin, H. and Bockisch, C., 2013. The Potential of Omniscient Debugging for Aspect-Oriented Programming Languages. In: *Proceedings of the 1st Workshop on Comprehension of Complex Systems*, CoCoS '13. [online] New York, NY, USA: Association for Computing Machinery. pp.13–16. https://doi.org/10.1145/2451592.2451597.

Yin, H., 2013. A graphical tool for observing state and behavioral changes at join points. In: *AOSD 2013 Companion - Proceedings of the 2013 ACM on Aspect-Oriented Software Development*. pp.29–30. https://doi.org/10.1145/2457392.2457405.