Aspect-Oriented Programming Revisited: Modern Approaches to Old Problems

1st Tim Träris
Faculty of Computer Science
Furtwangen University
Furtwangen, Germany
tim.jannes.traeris@hs-furtwangen.de

2nd Maxim Balsacq
Faculty of Computer Science
Furtwangen University
Furtwangen, Germany
maxim.balsacq@hs-furtwangen.de

3rd Leonhard Lerbs
Faculty of Computer Science
Furtwangen University
Furtwangen, Germany
l.lerbs@hs-furtwangen.de

***Abstract*—Although aspect-oriented programming has been around for two decades and is quite popular in academic circles, it has never seen broad adoption. For this reason, this work revisits the early approaches and identifies issues that may explain the lack of adoption outside academia. In this regard, we present different weaving mechanisms and showcase modern approaches to aspect-oriented programming which attempt to solve some of the issues of earlier approaches. Subsequently, we discuss difficulties of real-world uses and the gained paradoxical modularity. Finally, we present a set of issues to consider before choosing an AOP approach.**

***Index Terms*—Aspect-oriented programming, cross-cutting concern, modularity, aspect weaving**

I. Introduction

Aspect-oriented programming (AOP) is a programming paradigm aiming to provide generic functionality across multiple parts of the program in order to solve cross-cutting concerns and increase modularity. In the following, we introduce basic concepts and explain commonly used terminology.

A. Separation of Concerns

Software has become increasingly complex [1, 2], especially over the last two decades. Separation of Concerns, a term originally coined by Dijkstra [3], has been one generally agreed upon concept to tackle this issue. In modern terms, the basic idea intends to split a program into independent parts by grouping related functionality. As a result, code becomes easier to read, understand, maintain, extend, and reuse. However, depending on the level of abstraction, developers face hard to overcome challenges as separating concerns in practice is often easier said than done.

B. Cross-Cutting Concerns

Features like logging and security aspects of a program usually affect, interact with, or rely on other parts of the software. Such concerns cannot be assigned to a specific part or module of the software and therefore have to be implemented at multiple locations in the source code, effectively cross-cutting the program, hence the name. Inevitably, these fragments introduce either duplicate code, unnecessarily complex interdependencies, or both. As a result, cross-cutting concerns compromise the well-established and often striven for concept of modularity. Unfortunately, this is an issue neither procedural programming nor object-oriented programming (OOP) can solve in a clean way [4].

C. Introducing Aspects

Aspect-oriented programming intends to retain modularity by encapsulating cross-cutting concerns in so-called aspects. This enables developers to implement otherwise cross-cutting functionality separately from the core concerns. Aspects can alter core program behavior by applying changes, commonly referred to as advice. This happens at implicitly defined join points throughout the program depending on the exact implementation (e.g. when a method is called or an object is constructed). So-called pointcut queries define which advice is applied at which set of join points in the program. Figure 1 shows a visual representation of this process.

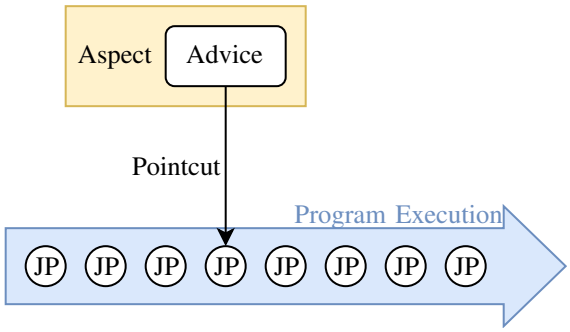

Fig. 1. Aspects provide advice which is applied via pointcut query at one or multiple join points (JP) during program execution

D. Weaving Aspects into Code

Allowing programmers to implement cross-cutting concerns as aspects implies that those detached parts must be combined later to form an executable program. This process is known as aspect weaving. Weaving may happen at different times, depending on the exact implementation of the aspect weaver. Exemplary, a simple implementation may generate code based on the aspect definition, which is injected into the source code before compilation. However, especially in bytecode languages like Java, aspect weaving is often integrated into the compilation process. Instead of generating woven code, aspects and classes are compiled into bytecode before weaving takes

place, usually resulting in more efficient program execution [5]. We will further explain and discuss the different weaving approaches in section IV.

E. AspectJ

AspectJ was the first practical implementation of AOP, initially developed by the AOP pioneers at Xerox. Today, it is available under the Common Public License and maintained by the Eclipse Foundation [6]. As such, it has influenced newer implementations. AspectJ is a commonly used aspect-oriented extension of the Java programming language [7]. To illustrate the basic concepts of AOP, listing 1 shows an exemplary aspect definition in AspectJ. In this example, we define a pointcut `methodCall()` which will be applied whenever the method `foo()` of class `DemoClass` is called. This pointcut provides one advice to be executed before the method is called. The advice will print a logging message regarding the method call in our example.

Listing 1
ASPECT EXAMPLE IN ASPECTJ

```
public class DemoClass {
    public void foo() {
        System.out.println("This is foo()!");
    }
}

public aspect DemoAspect {
    pointcut methodCall():
        call(* DemoClass.foo(..));

    before(): methodCall() {
        System.out.println("foo() called!");
    }
}
```

While AspectJ mainly supports compile-time weaving (see section IV-B), it also supports load-time weaving [8]. As we will show later, most modern approaches prefer compile-time weaving and still provide similar pointcut selection features as AspectJ.

II. RELATED WORK

In the paper Aspect-Oriented Programming is Quantification and Obliviousness [9], Filman and Friedman try to identify AOP and fundamentally describe its properties. As the title suggests, they loosely define AOP as a combination of quantification and obliviousness. The former means that one aspect can apply advice at multiple join points. Thus, aspects enable programming statements expressing the following:

In programs P, whenever condition C arises, perform action A. (1)

The authors argue that this quantification must be either conducted statically (onto the program structure) at compile-time or dynamically tied to an event happening at run-time (e.g. due to exceptions, call stack and program history).

Obliviousness refers to the fact that core concerns have no way of knowing which advice is applied at which join points. Therefore, program P is oblivious to which actions A are performed.

In the further course of the paper, Filman and Friedman bring attention to multiple design decisions and open questions regarding an AOP systems behavior, composition, implementation, and development process. In particular, the authors point to the following issues:

- Different approaches: AOP frameworks can generally take two different approaches: Clear-box or black-box. The former quantifies over known program structure as both program and aspect internals can be examined. The latter on the other hand wraps aspects by prohibiting the view of the program and aspect. Therefore, quantification is carried out by using a public interface and declaring functions and methods while hiding internals.
- Structural features: AOP systems must provide an efficient way to implement the newly required structures. This includes a language to define aspects and pointcuts as well as rules specifying the scope of conditions C which may serve as join points.
- Interface composition: AOP systems need to reevaluate compositional questions i.e. provide an interface for actions A to interact with the base program. Thus, questions arise like: What privileges does an aspect have, especially compared to the main program? How are aspects visible to each other? Can aspects communicate with each other and are there mechanisms to resolve inconsistencies e.g. conflicts between aspects?
- Weaver implementation: There are many ways to include the AOP code into a program. Are aspects compiled separately? Are they applied to the source code or bytecode? At which point in time are aspects applied: Before, during compilation or at run-time?

The authors conclude among other things that, although most AOP implementations are based on an OOP language, AOP properties are not per se tied to an OOP language. In fact, while the AOP paradigm may benefit from some OOP circumstances, non-OOP languages can describe quantification and obliviousness as well.

We will confirm this statement as we revisit some of the presented questions by comparing different weaving mechanisms in section IV and subsequently showcasing modern AOP approaches in section V. To fully understand the need for modern AOP approaches, we examine the major issues of early AOP implementations in the following section.

III. SHORTCOMINGS OF THE ORIGINAL

The concept of AOP was introduced two decades ago. At that time OOP was already widely adopted. Some concerns regarding the structural cleanliness of OOP code emerged, mainly because cross-cutting concerns could not be avoided by the OOP style. OOP decomposes concerns into objects in order to implement them as separated as possible. Therefore, in an ideal world, cohesive objects form a solution for one - and only one - concern [10]. Unfortunately, limitations of that strategy are quickly reached when trying to separate some

concerns into objects due to the inability to correctly localize concerns and adapting those to the OOP style [11]. While the emerging AOP style looked promising to solve some of these limitations [4], we can identify several shortcomings today.

A. Historic View

In the beginning of AOP, one could only find simple examples on where to use it. Those examples mostly consisted of some sort of logging, tracing or debugging problems. What seemed to be an analogy to widely known OOP examples like derivation (e.g. Vehicle - Car; Vehicle - Bus), turned out to be a problem which could be resolved by using tools provided by an IDE (i.e. tracing), introducing new features (e.g. error handling) or using different, more domain-specific languages to address the problem (e.g. database transaction management) [12]. Alongside some other common examples like a display updating aspect, those examples are widely used today to describe what AOP is good for [12]–[14].

Further, the AOP concept has been implemented to varying degrees depending on the language and objectives [4]. Numerous implementations for different languages and frameworks have been created. Many of them use only a subset of the initial theoretical concept of AOP described by Kiczales [4, 15].

B. Debugging Becomes Challenging

As described in section II, AOP is often loosely defined as the combination of quantification and obliviousness [9]. While both of these properties enable the separation of cross-cutting concerns, both cause significant difficulties when reading, understanding and debugging code. In fact, AOP can heavily obscure the control flow of a program. Since aspect awareness is ruled out per AOP's definition, the exact order of programming statements becomes incomprehensible to a programmer looking just at the base code. As a result, the overall readability of the program's source code as well as its maintainability is severely hampered. Naturally, debugging and fixing programming error becomes a complex challenge. This is especially concerning when considering the fact that on average, programmers spend almost half of their time doing debugging tasks [16]. Besides, obliviousness may impose obstacles (e.g. longer familiarization time) in the on-boarding process for new programmers joining the team.

Further, shared characteristics between AOP pointcuts and the infamous GOTO statement can be identified [17]. This statement performs a one-way jump in the execution logic of a program without returning to its origin like a conventional method call would. Dijkstra criticized this behavior [18] arguing that programmers must always be able to comprehend the value of variables at certain points (coordinates). Similar to GOTO, pointcuts can have a major impact on control flow and their obscurity makes it virtually impossible to create a reasonable coordinate system based on the programming language alone.

C. Unforeseen Consequences of Aspect Modularity

The problems with obliviousness and quantification grow even further as the number of aspects increases. Like the base code, aspects themselves are also unaware of other aspects defined elsewhere. As a result, multiple aspects may apply advice to the same join points. While this provides opportunities for efficient concern separation, it may also involve unforeseen complications and inadvertent interactions between aspects. For example, aspect A may depend on a certain result of a method which has been modified due to the behavior of aspect B. While programming, the correlation between aspect A and B is obscured. Therefore, a developer writing aspects must have a thorough understanding of the program and all other aspects in order to assess how they interrelate.

Additionally, aspects can provide advice applying to themselves and therefore modify their behaviour unless not explicitly prevented by their definition or the AOP framework [19]. This introduces major recursion risks as well as antinomy issues like the classical liar's paradox. The antinomy of the liar is a logical problem arising when a sentence states its falsehood. For example, if the statement 'This sentence is false.' is false, its assertion is in fact true and vice versa. The statement therefore dissents itself due to the self reference. Conferred to AOP, this means that aspects could counter each other and prevent each other from applying advice.

Further, many AspectJ-like AOP implementations strictly distinguish between regular parts of the program (advised code) and aspects (advising code). Aspects can advise classes as well as other aspects using the `adviceexecution` pointcut [10]. This is referred to as an asymmetrical AOP implementation [20] since it is not true the other way around (regular classes cannot advise aspect definitions). When compared to symmetric implementations, asymmetric implementations offer powerful opportunities for cross-cutting concern definitions. However, some argue that sacrificing symmetry for AOP is unnecessary as symmetric approaches are capable of expressing cross-cutting behavior while preserving orthogonality and the reusability of components [20]–[22].

D. Concurrency and Thread Safety

The issues discussed so far have an even greater impact when developing multi-threaded applications. Many programming languages implement thread-safe objects and methods differently, as they usually introduce a performance penalty compared to non-thread-safe versions of the same code. With this in mind, obliviousness facilitates yet another problem: If the aspect's code is not written in a thread-safe manner, it may inadvertently modify otherwise thread-safe code of the base program. Consequentially, thread safety may be lost, possibly resulting in data corruption and undefined behavior of the application.

E. Refactoring Breaks Pointcuts

As shown previously in listing 1, aspect definitions require detailed pointcut statements including class and method

names in many AOP implementations [23]. Renaming and deleting classes and methods in the program can therefore render advice inapplicable when the aspect definition is left unchanged. This is often referred to as pointcut fragility. Since the base code contains no trace of applied aspects due to the obliviousness concept of AOP, developers have no direct way of knowing whether their change requires a change in an aspect definition as well. As a result, it is vital that developers agree on naming conventions and must rely on IDE support in order to stay on top of things.

F. Performance

AOP performance heavily depends on the used AOP framework and its aspect weaver implementation. In a systematic literature review of AOP weavers [13], Soares et al. found that the number of existing experiments in literature regarding AOP weaving is insufficient to come to a conclusion about performance. The authors then decided to perform their own experiment based on the knowledge of the review to test AOP performance with AspectJ.

For the experiment, they took the time, CPU load, and memory consumption as measurement indicators and used compile-time weaving, load-time weaving, and run-time weaving in comparison with a non-AOP approach. The results indicated that compile-time weaving had no significant impact on the measurements. Surprisingly, the CPU performance was about 8% less compared to non-AOP. Having a load-time weaving approach used one third more memory and having a run-time weaver used about 40% more time and more memory. Both load and run-time weaving had slightly better CPU performances than the normal non-AOP approach [13].

IV. Weaving Models

The presented frameworks use different weaving techniques to apply the advice of AOP aspects to code. Depending on the weaving implementation, multiple advantages and disadvantages must be considered. In this section, we will illustrate the impact of run-time, compile-time, and load-time aspect weaving.

A. Run-time Weaving

When code weaving is done at run-time, the weaver must have the ability to determine where pointcuts should be placed. Therefore, a way to locate functions and manipulate them is required. In some languages running inside a virtual machine, e.g. Java, reflection enables access to and manipulation of this information using the meta-model of the programming language. This allows making changes to the cross-cutting logic at run-time without recompiling the application [24]. As we present later in section V-C, the Spring AOP framework uses this approach. However, system programming languages such as C, C++, or Rust usually do not preserve such information during compilation. This makes it more difficult or even impossible to use run-time code weaving. In general, the ability to perform run-time code weaving is limited by which information can be obtained at run-time.

B. Compile-time Weaving

Weaving code at compile-time implies that to apply advice, the existing code must be parsed and transformed by the weaver. Therefore, the weaver must either parse and transform the code itself or be integrated into existing compiler/parser tools. For example, the C/C++ weaver implementation for LARA (named CLAVA [25]) uses the latter approach: It is integrated into clang, enabling it to access any information a compiler could access.

There are multiple advantages to compile-time aspect weaving:

- The full code is accessible to the weaver and may be manipulated using AOP.
- The weaved code (which includes the AOP advice) may be optimized by the compiler.
- The AOP code may choose optimizations the compiler should apply.
- In case of source-to-source compilation, static analysis of the generated code is possible.

However, compile-time weaving also has the following disadvantages:

- Complex additional tooling is required for handling the source code transformation. If this tooling relies on compiler/parser internals, it may lead to a vendor lock-in if not enough alternatives are available. On the other hand, if the parser is hand-written, it may go out of date and not support newer language features.
- Parts of the source code required to build an application may not be available.

C. Load-time Weaving

Although the two weaving methods presented above may be sufficient for most use cases, neither is optimal for some use cases e.g. the Open Services Gateway initiative (OSGi) platform. The OSGi presents a framework to provide a component model in a service-oriented architecture where components can dynamically be installed, updated, started and stopped, without the need for a reboot [26].

There is no point in using compile-time code weaving due to the nature of OSGi. Using run-time code weaving within the OSGi framework is not possible due to the class visibility of a bundle. Each bundle supplies its own class loader that can be accessed through the service registry. However, the bundle which is accessed needs to be known during compile-time. This means that a bundle using aspects cannot be modified at run-time, only when loading, thus introducing load-time code weaving [26].

D. Summary

AOP requires an aspect weaver that is familiar with the target programming language. This raises tooling and performance requirements either at compile-time (due to required compiler tooling) or run-time (due to code modification at load-/run-time). The information available to weavers limits what can be done in aspects.

V. Modern Approaches

As with OOP implementations, a lot of different AOP frameworks exist deviating in their way of distributing join points, applying pointcuts, and defining aspects. In fact, many AspectJ-like implementations are built on top of an existing OOP systems. In this chapter, we showcase modern approaches and AOP implementations attempting to solve some of the original shortcomings discussed previously.

A. LARA

LARA is an aspect-oriented programming language originally designed for embedded systems [27]. It allows granular access to code based on a language-independent abstract syntax tree (AST).

An example tree of a subset of a common language specification is shown in figure 2. Using an AST means that pointcuts are not limited to functions but can be applied to variables, loops, classes, and other structural features. These pointcuts may be joined to create an even further refined selection. The properties of the AST can be examined to filter pointcuts by their properties. As a result, LARA is not limited to classes or function calls and may even be used in non-OOP languages (see II). In general, the LARA domain-specific language (DSL) allows more granular access to pointcuts than other AOP implementations [28].

Implementation of the LARA concept is done by supplying a weaver for the target language. We found LARA weaver implementations for MATLAB, C/C++, Java, and JavaScript [29]. When implementing a weaving tool, the LARA code is source-to-source translated to the target language. This means that the same advice can be applied to different target languages. With this in mind, LARA weavers inherit the advantages and disadvantages of compile-time code weaving.

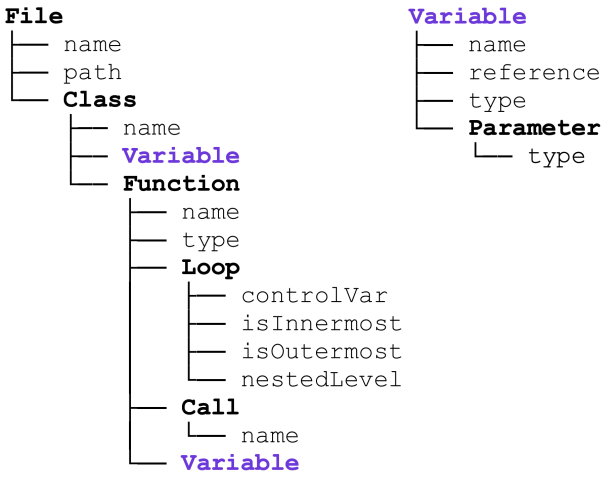

Fig. 2. Tree-like representation of an exemplary common language specification subset [27, 28].

Since the AST is language-independent, an issue arises: How does code insertion work? LARA solves this issue by providing a simple language including standard functions (such as timers and printing to standard output) that compiles down to whatever language the underlying code uses [28]. Since the LARA DSL itself is independent of the used languages, the AOP code can easily be applied to multiple languages.

Beyond code insertion, LARA also allows for easy control of various language-specific optimizations. For example, CLAVA (the LARA weaver implementation for C/C++ [25]) allows unrolling loops selected by an AOP pointcut. This unrolling reduces looping overhead and may improve performance. Types of variables may also be altered (such as reducing the precision of floating-point variables) to improve performance at cost of precision.

B. GAMESPECT

GAMESPECT aims to provide a meta-level DSL initially made for the video game engine Unreal Engine 4 (UE4) [30]. With this engine, it is possible to run scripts in different languages to perform game-specific tasks - a feature intending to offer a low barrier entry for new developers. Those languages include C++, Lua, Skookum, and Blueprints (an internal GUI-based editor for gameplay mechanics). Like many game engines, UE4 offers a built-in messaging system to send messages or events to other 3D objects. This creates a secondary dependency structure. Classes cannot only derive from each other but they can also be contacted in a 3D context, e.g. a player class can derive from an entity class. In the 3D space, a player is a member of the game scene. Whenever a player interacts with another player, they contact each other in a 3D context via systems supplied by the game engine. This is important because it naturally breaks many standard code patterns and requires a different way of thinking when developing code in such engines [31, 32].

The developers of GAMESPECT saw potential in the use of AOP for game development because similar tasks are often performed at different locations in the code. This is especially true for UE4, as it layers a script based ecosystem on top of the actual game engine.

GAMESPECT's primary focus lies on join points. It requires only one weaver which supplies platform-independent implementations. This is achieved by defining GAMESPECT as a meta-DSL based on the UE4 DSL, thus decoupling GAMESPECT from the lower level DSL of UE4. This separation is important because UE4 supports multiple languages and code can therefore be generated for AspectC++, Skookum, and Blueprints. Furthermore it enables GAMESPECT to not only work in the UE4 context but with other game engines as well.

GAMESPECT was inspired by LARA in how it supports multiple languages in source-to-source compilation. To solve the formal composition specification as described by Mishali and Lorenz, GAMESPECT applies the work from its predecessor SPECTACKLE [33]. SPECTACKLE's authors found a common language specification subset to the languages used by the game engine [30].

In the GAMESPECT paper, Geisler et al have shown that GAMESPECT could shorten the amount of code written by game developers on found aspects from 9% up to 40% while maintaining efficiency and modularity with concurrent support for all three target languages Skookum, C++ and Blueprints [30].

C. Spring

Spring AOP is part of the Java Spring Framework. It performs run-time weaving, relying on Java reflection to determine pointcuts and proxy classes to apply the advice code [34]. As shown in figure 3, the class of the original code is replaced by a proxy that implements all the methods of the original class. By inserting the advice code into the proxy object, the advice code can be executed before or after a call to the original function. This allows to trace and/or modify the result of the function call.

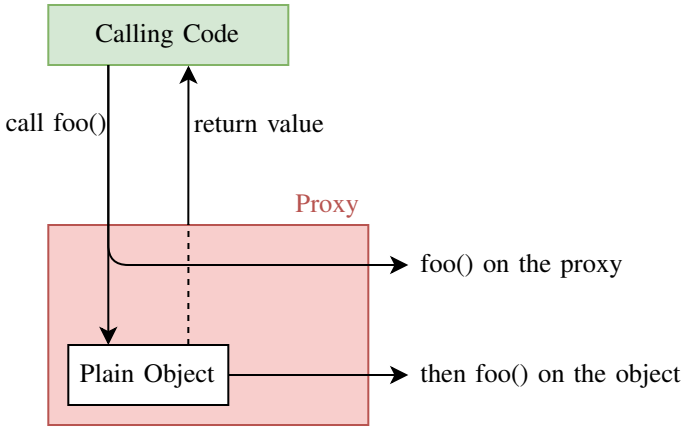

Fig. 3. The proxy pattern as used by Spring AOP [34]

Using reflection and the proxy pattern has various advantages and disadvantages: The main advantage is that no extra tooling is required, resulting in portable, easy, and fast compilation. Solely using the default Java compiler, pointcuts and advice can be created and used. This means that AOP can also be used on files where no source is available (e.g. when using third party libraries) or compilation is not desirable (e.g. the standard library).

However the use of reflection introduces a few major drawbacks, some based on technical limitations of the Java reflection API. One such technical limitation is that the parameter names are not available and must be specified by the user [34], introducing potential for errors.

Another technical limitation is that code can only be hooked on a per-function or per-class field basis. As a result, any methods containing large amounts of code may not be altered at the required level of granularity. Other languages like LARA or GAMESPECT sidestep this issue by using source-to-source compilation and defining features of the AST as join points.

Since Spring AOP uses reflection, the code can only be modified at run-time, meaning that it inherits all the advantages and disadvantages of run-time code weaving.

In general terms, a limitation of any framework using reflection is that the cost of introducing AOP is moved from compile-time to run-time. This applies to performance e.g. setup time for reflection, as well as possible failures e.g. in case a library is updated and changes its internal API.

VI. Discussion

In this section, we take a critical look at AOP and its use cases as well as the ambitious goal it tries to solve. In this regard, we also reflect on its popularity and discuss the paradoxical problem of modularity. Finally, we review the performance impact of AOP.

A. An Overambitious Goal?

As introduced in III, the example use cases that are usually given for aspects do not show the whole potential of AOP. They are usually limited to a set or type of aspects that reoccur between different papers. When looking at a real-world example, as shown by the GAMESPECT authors, the results are sobering. The cross-cutting concerns found in a real-world game originally consisted of 288 lines of code. After converting them to aspects, the amount of lines used to implement cross-cutting concerns could be reduced to 223, thus saving one-third of the lines. However, this involves only code regarding cross-cutting concerns, which represents only a marginal fraction compared to the whole game code base.

Although a game engine has a lot of potential for AOP, the examples shown by Geisler et al. in the GAMESPECT paper could also be implemented using the internal messaging system of UE4. This system provides the ability to fire global events or using the 3D scene hierarchy to contact other objects. In this case, relinquishing AOP not only makes the code more readable and understandable, but also results in not working around the built-in possibilities of UE4 by unnecessarily introducing a higher level DSL.

Another example is the use of AspectJ during refactoring of the Berkeley DB project [14]. The authors found that it is difficult to maintain the original crafted concept of externally available interfaces while introducing advice and pointcuts to Berkeley DB. They often found themselves in a position where they needed to either write method hooks or lower the scope of methods to public, thus compromising the public/private concept of Berkeley. Further, AspectJ is not able to alter method signatures. Since exceptions require a throws declaration in the method signature, aspects cannot add new exceptions to methods.

B. An Abating Hype?

We cannot fail to be impressed by the velocity at which AOP gained traction after its emergence two decades ago. Since OOP, aspect-oriented programming is one of the most popular programming paradigms in academia. This even led to multiple editions of AOSD, an international scientific conference solely focusing on modularity and AOP, being held annually from 2002 until 2015 [35]. However, it is still difficult to speak of AOP as a general success and *the* new way of programming

- a viewpoint the AOP hype might suggest. In fact, Google Trends data for interest in the search term 'aspect oriented programming' shown in figure 4 paints a different picture. Although proprietary analytical data must be taken with a grain of salt, the graph clearly shows the decline of interest in AOP since its hype in the early years of this century.

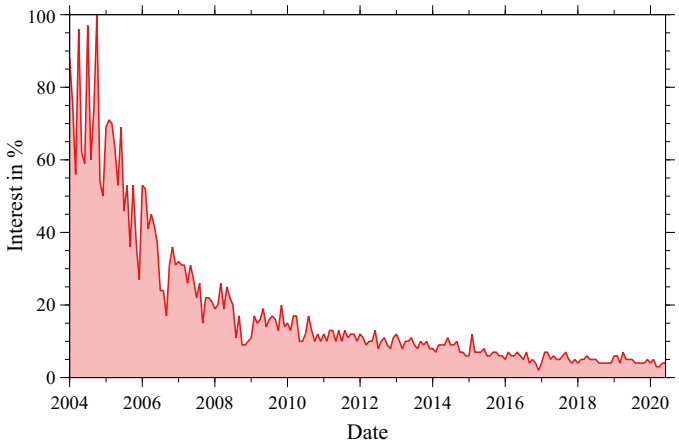

Fig. 4. Global interest in the search term '*aspect oriented programming*' according to Google Trends [36]. 100% represents the highest amount of interest within the time period.

One possible explanation for this lies in the nature of academic research (sometimes cynically) criticized by its members over the years [12, 37, 38]. Research projects in computer science usually depend on funding and thus face pressure to produce promising innovations and new ideas. AOP ticks a lot of boxes in this regard, as it potentially offers a way to solve cross-cutting concerns by introducing new structures for code composition [39]. With this in mind, the early popularity of AOP among the research community is hardly surprising.

C. The Paradox of Modularity

Aspects are often implicitly yet strongly coupled to their respective target join points. This is concerning as it poses a potential threat to modularity [12], an issue which AOP initially tries to improve.

In OOP, object modularity is achieved by information hiding and data encapsulation. Ideally, objects group data and all methods operating on that data in order to restrict access. Communication between modules is implemented by providing interfaces, i.e. specification of variables and methods, without revealing their implementation.

AOP aims to bring modularity to cross-cutting concerns by moving them into aspects. Paradoxically, this implies that aspects require access to parts of the program hidden in modules in order to apply advice according to their pointcuts. As a result, aspects need one of the following:

- An explicit interface between aspects and modules. Naturally, this interface must be updated whenever the module and its methods change. Subsequently, aspects must be modified as well to adopt changes of the interface.
- Implicit access to the module. This approach inherits the issues of the former while also leaving programmers unaware of the dependency. As a consequence, data and behavior which seemed to be encapsulated by the module might change when advice is applied.

Therefore, improving the modularity of cross-cutting concerns results in a loss of object modularity. At first, this may sound like a reasonable trade for some scenarios. However, section III-C proves the opposite. As soon as aspects interrelate, obtained modularity is lost in the same way. In conclusion, early claims in literature that AOP is able to 'modularize the un-modularizable' [40] have proven to be untenable [41].

D. Performance considerations

As described in section III-F, the type of weaver can have a significant impact on performance. However, the authors also argue that they were not able to find many common concerns that have been implemented by the reviewed papers [13]. Therefore, this performance study may not be representative.

Additionally, the results of performance studies may be susceptible to distortion. In all examples of Soares et al. as well as the refactoring of Berkeley DB, AOP was introduced to the project in a later stage. Building an AOP approach from the ground up, i.e. involving AOP in the specification and design phase of the project, may yield different results.

In fact, AOP may also help to improve performance. By instrumenting function calls (e.g. with timers), it allows to evaluate and improve existing code. Some AOP frameworks (such as LARA) may even directly help to optimize existing code by allowing fine-grained control over internal compiler optimizations. In general, further work is needed in order to investigate AOP performance impacts, especially in regards to the point of time where AOP is embraced.

VII. Conclusion

In this work, we revisited early AOP approaches and investigated multiple issues that possibly had an impact on the adoption of AOP. Based on the gained insight, we analyzed modern AOP implementations to learn how they approach cross-cutting concerns and aim to solve the gathered issues. We found that LARA and GAMESPECT directly access the AST of the programming language while Spring uses reflection to facilitate run-time code weaving. Both of these techniques enable a more detailed access to information compared to the early AspectJ approach. As a result, they allow for more accurate placement of pointcuts, reducing the risk of pointcut fragility and mitigating some modularity problems.

However, even the presented approaches fail to solve some of the identified issues. This is hardly surprising, given that some issues, such as the paradox of modularity and the liar's problem are unsolvable, even in theory.

During our research, we found many hints about the exemplary use of AOP in the context of simple use cases like tracing or logging. Nevertheless, attempted uses in more complex scenarios lead us to the conclusion that aspects, especially

with inter-dependencies on each other, do not scale well and are difficult to implement and maintain.

Finally, the following issues must be carefully considered before choosing an AOP approach:

- Is AOP used to solve simple use cases or complex use cases? Simple use cases introduce a relatively high additional workload in order to add aspect systems and weavers to the project, whereas complex use cases bring along unwanted inter-aspect side effects due to coupled aspects.
- Is performance a limiting factor? Choosing an AOP approach may have a significant impact on the performance of the applications. The exact impact must be investigated further since we could not find a representative analysis that compared simple and complex use cases.
- How will AOP impact the project architecture? Introducing AOP later in the project might destroy already defined structural modifiers of the application. On the other hand, introducing AOP in the specification stage results in increased planning overhead. Further, AOP cannot easily be removed later.
- Are there any existing technologies on the same domain language level that may achieve this behavior? Introducing a higher level domain might be unnecessary and increases the complexity of the application.
- What abilities does the weaver have? Its abilities will limit how AOP can be used, possibly requiring modifications to the architecture of the application code.

REFERENCES

[1] R. N. Charette, "This car runs on code," *IEEE spectrum*, vol. 46, no. 3, p. 3, 2009.

[2] E. E. Ogheneovo, "On the relationship between software complexity and maintenance costs," *Journal of Computer and Communications*, vol. 2, no. 14, p. 1, 2014.

[3] E. W. Dijkstra, "On the role of scientific thought," in *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 60–66.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European conference on object-oriented programming*. Springer, 1997, pp. 220–242.

[5] E. Hilsdale and J. Hugunin, "Advice weaving in AspectJ," in *Proceedings of the 3rd international conference on Aspect-oriented software development*, 2004, pp. 26–35.

[6] Eclipse Foundation. The AspectJ Project. [Online]. Available: https://www.eclipse.org/aspectj/

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *European Conference on Object-Oriented Programming*. Springer, 2001, pp. 327–354.

[8] AspectJ Developers. AspectJ load-time weaving. [Online]. Available: https://www.eclipse.org/aspectj/doc/released/devguide/ltw.html

[9] R. E. Filman and D. P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," in *Workshop on Advanced Separation of Concerns*, vol. 2000, 2000.

[10] H. Rebêlo and G. T. Leavens, "Aspect-oriented programming reloaded," in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, 2017, pp. 1–8.

[11] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Communications of the ACM*, vol. 44, no. 10, pp. 29–32, 2001.

[12] F. Steimann, "The paradoxical success of aspect-oriented programming," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006, pp. 481–497.

[13] M. S. Soares, M. A. Maia, and R. F. Silva, "Performance Evaluation of Aspect-Oriented Programming Weavers," in *International Conference on Enterprise Information Systems*. Springer, 2014, pp. 187–203.

[14] C. Kastner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *11th International Software Product Line Conference (SPLC 2007)*. IEEE, 2007, pp. 223–232.

[15] W. Golubski, *Entwicklung verteilter Anwendungen: Mit Spring Boot & Co*. Springer-Verlag, 2020.

[16] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[17] C. Constantinides, T. Skotiniotis, and M. Stoerzer, "AOP considered harmful," in *1st European Interactive Workshop on Aspect Systems (EIWAS)*, 2004.

[18] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[19] F. Forster and F. Steimann, "AOP and the antinomy of the liar," in *Workshop on the Foundations of Aspect-Oriented Languages (FOAL) at AOSD*. Citeseer, 2006, pp. 47–56.

[20] W. Harrison, H. Ossher, P. Tarr, and W. Harrison, "Asymmetrically vs. symmetrically organized paradigms for software composition," *IBM Rsch. Rpt. RC22685 (W0212-147)*, 2002.

[21] S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto, "Do we really need to extend syntax for advanced modularity?" in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, 2012, pp. 95–106.

[22] H. Rajan and K. J. Sullivan, "Unifying aspect-and object-oriented design," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 19, no. 1, pp. 1–41, 2009.

[23] K. Gybels and J. Brichau, "Arranging language features for more robust pattern-based crosscuts," in *Proceedings of the 2nd international conference on Aspect-oriented software development*, 2003, pp. 60–69.

[24] C. Schaefer, C. Ho, and R. Harrop, "Introducing Spring AOP," in *Pro Spring*. Springer, 2014, pp. 161–239.

[25] J. Bispo. Clava: C/C++ Source-to-Source Tool based on Clang. [Online]. Available: https://github.com/specs-feup/clava

[26] T. Keuler and Y. Kornev, "A light-weight load-time weaving approach for OSGi," in *Proceedings of the 2008 workshop on Next generation aspect oriented middleware*, 2008, pp. 6–10.

[27] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: an aspect-oriented programming language for embedded systems," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, 2012, pp. 179–190.

[28] P. Pinto, T. Carvalho, J. Bispo, M. A. Ramalho, and J. M. Cardoso, "Aspect composition for multiple target languages using LARA," *Computer Languages, Systems & Structures*, vol. 53, pp. 1–26, 2018.

[29] J. Bispo and T. Carvalho. LARA implementations. [Online]. Available: http://specs.fe.up.pt/index.php?page=tools

[30] B. J. Geisler, F. J. Mitropoulos, and S. Kavage, "GAMESPECT: Aspect Oriented Programming for a Video Game Engine using Meta-languages," in *2019 SoutheastCon*. IEEE, 2019, pp. 1–8.

[31] E. Games. IMessageBus. [Online]. Available: https://docs.unrealengine.com/en-US/API/Runtime/Messaging/IMessageBus/index.html

[32] U. Technologies. IMessageBus. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/MessagingSystem.html

[33] D. H. Lorenz and O. Mishali, "SPECTACKLE: toward a specification-based DSAL composition process," in *Proceedings of the seventh workshop on Domain-Specific Aspect Languages*, 2012, pp. 9–14.

[34] Spring AOP developers. Spring API documentation. [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-advice

[35] AOSD. AOSD Conference 2002-2015 On Modularity. [Online]. Available: http://aosd.net/conference/

[36] Google. Google Trends aspect oriented programming. [Online]. Available: https://trends.google.com/trends/explore?date=all&q=aspect%20oriented%20programming

[37] T. Caulfield and C. Condit, "Science and the sources of hype," *Public Health Genomics*, vol. 15, no. 3-4, pp. 209–217, 2012.

[38] Z. Master and D. B. Resnik, "Hype and public trust in science," *Science and engineering ethics*, vol. 19, no. 2, pp. 321–335, 2013.

[39] H. Masuhara and G. Kiczales, "Modeling crosscutting in aspect-oriented mechanisms," in *European Conference on Object-Oriented Programming*. Springer, 2003, pp. 2–28.
[40] N. Lesiecki, "Improve modularity with aspect-oriented programming," *IBM DeveloperWorks*, 2002.
[41] A. Przybyłek, "Where the truth lies: Aop and its impact on software modularity," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2011, pp. 447–461.