Replicating AspectJ Constructs in Python: Limitations and Considerations

Bc. Viktor Modroczký

Institute of Informatics, Information Systems and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava

2023

Abstract

AspectJ is a widely adopted aspect-oriented language known for its extensive set of features. It enables the management of cross-cutting concerns in Java-based applications, without directly modifying their code. Although AspectJ is considered to be one of the best in the field, according to the 2023 Developer Survey conducted by Stack Overflow, Java is dominated by Python in popularity. On account of that, we take a look at what Python has to offer in the context of aspect-oriented software development. Without utilizing any third-party libraries, Python provides features, such as decorators and metaclasses, that can be used to implement aspect-oriented capabilities. However, using Python's decorators and metaclasses to replicate AspectJ advice can pose challenges and limitations that we explore in this paper.

1 Introduction

Aspect-oriented programming offers a way of resolving cross-cutting concerns in applications without the need to modify existing code. AspectJ is the leading language in the aspect-oriented paradigm of software development, serving as an extension to Java [1]. Its practicality lies in so-called advice that provide means of adding functionality within join points in code execution [2].

In this paper, we discuss the features of Python, namely decorators and metaclasses, which can be utilized to adapt the aspect-oriented paradigm to the language by monkey-patching [3, 4]. Decorators allow one to extend the behavior of existing functions without modifying their source code explicitly. They are syntactic sugar for functions that take another function as a parameter, wrapping it and modifying its behavior [4]. Metaclasses define

the creation of other classes and, in addition to that, can manipulate their attributes and methods [4, 5].

Aspects in AspectJ, which comprise pointcuts, `before()` and `after()` advice, as well as `around()` advice, are woven into join points at the bytecode level before execution [6]. In contrast to Java, Python lacks an aspect-oriented extension similar to AspectJ. Python is a dynamic interpreted language; therefore, any decorator and metaclass used for aspect-orientation purposes is evaluated at runtime [7].

Moreover, Python not having an AspectJ-esque extension reduces the complexity of aspect-oriented solutions within the language. Considering how well-established and developed AspectJ is, we set out to explore limitations of using decorators and metaclasses to replicate AspectJ advice in pure Python. These limitations can take various forms, such as challenges in implementation and the scope of AspectJ features, rendering replication of its advice in Python a demanding task.

2 Introduction to AspectJ

Kiczales et al. [8] recognized that object-oriented and procedural programming languages often struggle to clearly capture some of the important design decisions that a program must implement. Consequently, the implementation of such design decisions becomes scattered throughout the codebase, resulting in difficult code maintenance. Properties that are addressed by these design decisions are referred to as aspects that cross-cut the system's core. Kiczales et al. [8, 9] introduced the concept of aspect-oriented programming, which presents a clear expression of aspects in a modular manner and serves as the foundation for the development of AspectJ.

2.1 AspectJ Concepts

AspectJ supports two types of cross-cutting implementation, dynamic and static. According to Kiczales et al. [9], dynamic cross-cutting allows additional implementation to run at certain points in the execution of a program, whereas static cross-cutting allows the definition of new operations on existing types. Dynamic cross-cutting constructs are pointcuts and advice, whereas static cross-cutting constructs are inter-type declarations.

The following sections provide an explanation of the essential concepts within the aspect-oriented programming paradigm, as well as AspectJ itself.

2.1.1 Join Points & Pointcuts

Join points are a crucial part of any aspect-oriented language. As stated by Clement et al. [10], the join point is a well-defined point in the execution of a program. Join points are captured by matching pointcuts, allowing for the

insertion of advice into captured join points. A single pointcut can capture any number of join points in the program flow.

As specified by Clement et al. [10], AspectJ pointcuts have access to five types of join points:

- Method or constructor call
- Method, constructor, or advice execution
- Access to or update of a field
- Initialization of classes and objects
- Exception handler execution

2.1.2 Advice

Advice are constructs that define what is going to be done when a pointcut matches a join point. There are three types of advice in AspectJ described by The AspectJ Team [11] and Clement et al. [10]:

- `before()` advice that run just as a join point is reached, before the program proceeds with it
- `after()` advice that run after the program proceeds with a join point
- `around()` advice that runs as a join point is reached, but has explicit control whether the program proceeds with a join point, and the ability to change values within the join point, unlike `before()` or `after()` advice

2.1.3 Inter-type Declarations

Inter-type declarations, alternatively referred to as introductions, are described by Laddad [6] as constructs that can be utilized to alter the static structure of classes, interfaces, and aspects in a program. Introductions can modify a class or a set of classes by adding new attributes and methods to them, and can also influence the class hierarchy within a program, as stated by Gradecki and Lesiecki [12].

2.1.4 Aspects

Aspects in AspectJ are described by Clement et al. [10] as units of modularity, encapsulation, and, through their meaningful naming, abstraction. They are designed to handle cross-cutting concerns by piecing pointcuts, advice, and introductions together.

3 AOP in Python

According to the 2023 Developer Survey conducted by Stack Overflow [13], Python is a more popular language than Java. It is therefore a viable option to explore Python as a language for replicating key AspectJ functionality.

To operate as an aspect-oriented programming language, Python does not require the use of third-party libraries. Its intrinsic features are suitable for managing cross-cutting concerns, although not as fully fledged as in Java-based applications using AspectJ.

Python is a dynamic language by nature, which allows for aspect orientation using a technique called monkey-patching [3]. Ramalho [4] describes monkey-patching as dynamically changing a module, class, or function at runtime, to add features or fix bugs.

3.1 Decorators

Python decorators are based on the decorator pattern, which, as defined by Gamma et al. [14], has the following intent:

> *"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."*

As explained by Sarcar [15], the decorator pattern is also called the wrapper pattern because in programming we often use wrappers to dynamically add functionalities. A key characteristic of the decorator pattern in the context of aspect-oriented programming is that the existing structure of decorated code remains untouched, and therefore developers do not introduce bugs in existing code.

In Python, everything is an object, according to the official language reference. Van Rossum and the Python Development Team [16] state:

> *"Objects are Python's abstraction for data. All data in a Python program are represented by objects or by relations between objects."*

Therefore, in Python, the decorator pattern can be applied to functions as well as to classes and their methods.

A simple decorator can be defined as follows:

```
1 def decorator(func: Callable[..., Any]):
2     def wrapper():
3         print("Running wrapper().")
```

```
4
5       return wrapper
6
7
8   @decorator
9   def decorated():
10      print("Running decorated().")
```

The syntax using the *at* symbol (@) is equivalent to the following explicit call to the decorator:

```
decorated = decorator(decorated)
```

Running `decorated()` will result in the output "Running wrapper().", which means that the decorator replaces the decorated function. To retain what the decorated function does, we need to define the decorator as such:

```
def decorator(func: Callable[..., Any]):
    def wrapper(*args, **kwargs):
        print("Running wrapper().")
        func(*args, **kwargs)

    return wrapper
```

The `*args` and `**kwargs` parameters are essential if the decorated function has any arguments or keyword arguments. Running `decorated()` will now result in the following output:

```
Running wrapper().
Running decorated().
```

However, if we wanted to create parameterized decorators, we would have to define them differently. As noted by Ramalho [4], parameterized decorators are not strictly decorators themselves; they are more like decorator factories that return the actual decorators. The following is an example of such a factory:

```
def decorator_factory(param1: int, param2: None | str = None):
    def decorator(func: Callable[..., Any]):
```

```
        def wrapper(*args, **kwargs):
            print("Running wrapper().")
            print(f"Doing something with {param1} and
            ↪  {param2}.")
            func(*args, **kwargs)

        return wrapper

    return decorator


@decorator_factory(7)
def decorated(x: Any, y=0.1, z="string"):
    print(f"Running decorated({x}, {y}, {z}).")
```

The output of running the function `decorated(("X", "Y"), y=0.7)` is then the following:

```
Running wrapper().
Doing something with 7 and None.
Running decorated(('X', 'Y'), 0.7, string).
```

Alternatively, as described by Ramalho [4], we can create a decorator or decorator factory as a class with the `__call__` method implemented, which offers both enhanced readability and conciseness. A class implementing this method has callable instances. The following is a decorator factory implementation using a class:

```
class DecoratorFactory:
    def __init__(self, param1: int, param2: None | str = None):
        self.param1 = param1
        self.param2 = param2

    def __call__(self, func: Callable[..., Any]):
        def wrapper(*args, **kwargs):
            print("Running wrapper().")
            print(f"Doing something with {self.param1} and
            ↪  {self.param2}.")
            func(*args, **kwargs)

        return wrapper
```

```
13
14
15 @DecoratorFactory(7)
16 def decorated(x: Any, y=0.1, z="string"):
17     print(f"Running decorated({x}, {y}, {z}).")
```

Running `decorated(["a", 0.5, 10], y=7.7, z="another string")` yields the following output:

```
1 Running wrapper().
2 Doing something with 7 and None.
3 Running decorated(['a', 0.5, 10], 7.7, another string).
```

Notice that using both `decorator_factory` and `DecoratorFactory`, we decorated a function with arguments and keyword arguments. These are passed to the original function `func` through the `wrapper` function in the `DecoratorFactory`'s `__call__` method and `decorator_factory`'s `decorator` function using `*args` and `**kwargs`.

3.2 Metaclasses

A metaclass in Python is a class that builds classes. One such class in Python is the `type` class. The `type` class being a metaclass means that every instance of it is a class, as stated by Ramalho [4].

According to Van Rossum and the Python Development Team [16], Python classes are created by the built-in `type` metaclass class if invoked with three arguments `type(name, bases, namespace)`, where `name` is the class name, `bases` is a tuple of base classes and `namespace` is a dictionary of name-value mappings of class methods and attributes.

Consider the following classes:

```
1 class A:
2     pass
3
4 class B(A):
5     text = "abc"
6
7     def double_text(self):
8         return self.text * 2
```

The same two classes can be created using the following `type` class invocations:

```
A = type("A", (), {})

B = type(
    "B",
    (A,),
    {
        "text": "abc",
        "double_text": lambda self: self.text * 2
    }
)
```

We can use `type` as a base class for custom metaclasses. Consider the following code:

```
class A(type):
    def __new__(cls, name, bases, namespace):
        def squared(self):
            return self.num**2

        # Alter class, e.g.
        namespace["num"] = 5
        namespace["squared"] = squared

        return super().__new__(cls, name, bases, namespace)


class B(metaclass=A):
    def __init__(self, num):
        self.num = num
```

Now, if we run the following code

```
b = B(10)

print(f"b.num from B.__init__ is {b.num}")
print(f"Class B has a squared method outputting {b.squared()}")
```

we get the following output:

```
b.num from B.__init__ is 10
Class B has a squared method outputting 100
```

In this way, we can alter classes that use our custom metaclass, by adding new attributes or methods to them.

4 Related Work

Kiczales et al. [9] proposed the usage of advice as a means of aspect-oriented programming. They introduced the AspectJ extension to Java that implements the proposed concepts. They describe the way in which AspectJ provides support for modular implementation of cross-cutting concerns, called aspects, and its interoperability with Java code. We use their work as a reference for implementing advice in Python.

Spinczyk, Gal, and Schröder-Preikschat [17] introduced an aspect-oriented extension to C++ that takes a similar approach to AOP as AspectJ. It also provides around, before, and after advice, and has pointcut definitions. AspectC++ uses compile-time weaving like AspectJ while introducing little to no run-time or memory overhead to application execution. Run-time overhead only occurs using certain pointcut functions that require runtime type checking.

Mărieș [18] implemented a Python library called aspectlib, which is an aspect-oriented programming library that uses monkey-patching and decorators. It provides a way to change the behavior of existing Python code. The implementation contains an `Aspect` decorator that can be used to decorate generator functions that yield advice. These generators can then be used to decorate functions whose behavior we wish to change.

Matusiak [19] in their work described various techniques and strategies of aspect-oriented programming in Python. They discussed the use of built-in Python features, such as decorators and metaclasses, for AOP. In addition to that, they listed and discussed libraries that use the aspect-oriented paradigm in Python. At the time of writing this paper, the aforementioned libraries are no longer available or maintained.

5 Implementation

In the previous sections, we clearly saw the aspect-oriented nature of Python's intrinsic features. Can we use them to recreate AspectJ constructs though? What are their limitations in this context?

To answer these questions, we will explore how these features can be leveraged to replicate key AspectJ constructs. Furthermore, we will identify

and discuss the practical limitations and challenges that may arise when attempting to implement such constructs.

However, AspectJ's feature set is vast, so to manage the scope of our analysis, we will need to focus on a specific subset of features used in our exploration of aspect-oriented programming in Python. We select a subset that contains only basic AspectJ features, such as simple `before()`, `after() returning`, `after() throwing`, and `around()` advice using simple pointcuts.

To implement these constructs, we utilize the latest Python release, which is version 3.12.0 at the time of writing. We choose the previously described features of Python, decorators and metaclasses, for implementation. We propose an implementation that uses monkey-patching, which is used to alter application execution during run-time, as a weaving technique. The implementation is available publicly on GitHub[1].

5.1 AspectJ Advice in Python

To implement AspectJ advice in Python, we used classes as decorator factories that take various parameters. The implemented advice are the following Python classes:

- `Before`
- `AfterReturning`
- `AfterThrowing`
- `Around`

5.1.1 Before

The `Before` class is the Python implementation of the `before()` advice. The constructor of this class takes the following parameters:

- `params_update` which is either a dictionary of key-value pairs, where the key is the name of the parameter to update and the value is the new value of the parameter, or `None`, if no update is desired. This dictionary is used to update positional and keyword arguments by their name. If the key is not the name of an argument, the given key-value pair is ignored. The function, `mutate_params`, used to update the parameters, is documented in the project source code found in the GitHub repository.
- `action` which is the function to run before the advised function.

[1] `https://github.com/vktr274/aspectpy`

- `action_args` which are the positional arguments of the action.
- `action_kwargs` which are the keyword arguments of the action.

The following code is the implementation of the before advice:

```
class Before:
    def __init__(
        self,
        params_update: dict[str, Any] | None,
        action: Callable[..., Any],
        *action_args,
        **action_kwargs,
    ):
        self.params_update = params_update
        self.action = action
        self.action_args = action_args
        self.action_kwargs = action_kwargs

    def __call__(self, func: Callable[..., Any]):
        @wraps(func)
        def wrapper(*args, **kwargs):
            args, kwargs = mutate_params(
                args, kwargs, self.params_update,
                ↪ signature(func)
            )
            self.action(*self.action_args,
            ↪ **self.action_kwargs)
            return func(*args, **kwargs)

        return wrapper
```

5.1.2 After Advice

The `AfterReturning` class is the Python implementation of the `after() returning` advice. The constructor of this class takes the following parameters:

- `params_update`
- `action` which is the function to run after the advised function returns. The action must have a parameter named `_RETURNED_VAL_` that represents the value returned from the advised function. The action must be decorated using `@validate_after_returning_action` which

checks whether the action follows the requirement. This decorator is further documented in the project source code found in the GitHub repository.

- `action_args`
- `action_kwargs`

The following code is the implementation of the after returning advice:

```
class AfterReturning:
    def __init__(
        self,
        params_update: dict[str, Any] | None,
        action: Callable[..., Any],
        *action_args,
        **action_kwargs,
    ):
        if not getattr(action, FLAG_VALIDATED, False):
            raise ValueError(
                f"{action.__qualname__} is not decorated "
                + f"with
                ↪  @{validate_after_returning_action.__name__}",
            )
        self.params_update = params_update
        self.action = action
        self.action_args = action_args
        self.action_kwargs = action_kwargs

    def __call__(self, func: Callable[..., Any]):
        @wraps(func)
        def wrapper(*args, **kwargs):
            args, kwargs = mutate_params(
                args, kwargs, self.params_update,
                ↪  signature(func)
            )
            result = func(*args, **kwargs)
            return self.action(result, *self.action_args,
            ↪  **self.action_kwargs)

        return wrapper
```

The `AfterThrowing` class is the Python implementation of the `after()`

`throwing` advice. The constructor of this class takes the following parameters:

- `params_update`
- `exceptions` which can be a class that inherits from the `Exception` class or a tuple of these classes or `None`. If it is `None`, all exceptions are caught, otherwise only stated exceptions get caught.
- `action` which is the function to run after the advice catches an exception based on the `exceptions` parameter.
- `action_args`
- `action_kwargs`

The following code is the implementation of the after throwing advice:

```
class AfterThrowing:
    def __init__(
        self,
        params_update: dict[str, Any] | None,
        exceptions: tuple[Type[Exception], ...] |
        ↪  Type[Exception] | None,
        action: Callable[..., Any],
        *action_args,
        **action_kwargs,
    ):
        self.params_update = params_update
        self.exceptions = exceptions or Exception
        self.action = action
        self.action_args = action_args
        self.action_kwargs = action_kwargs

    def __call__(self, func: Callable[..., Any]):
        @wraps(func)
        def wrapper(*args, **kwargs):
            args, kwargs = mutate_params(
                args, kwargs, self.params_update,
                ↪  signature(func)
            )
            try:
                return func(*args, **kwargs)
            except self.exceptions:
```

```
25                 return self.action(*self.action_args,
                   ↪  **self.action_kwargs)
26
27         return wrapper
```

5.1.3 Around Advice

The `Around` class is the Python implementation of the `around()` advice. The constructor of this class takes the following parameters:

- `params_update`
- `proceed` which can be a boolean or a function returning a boolean. This function should take a callable as a parameter, the advised function, so the advice can evaluate whether to proceed or not based on the signature or attributes of the advised function.
- `action` which is the function to run if the advised function does not proceed.
- `action_args`
- `action_kwargs`

The following code is the implementation of the around advice:

```
1  class Around:
2      def __init__(
3          self,
4          params_update: dict[str, Any] | None,
5          proceed: bool | Callable[[Callable[..., Any]], bool],
6          action: Callable[..., Any],
7          *action_args,
8          **action_kwargs,
9      ):
10         self.params_update = params_update
11         self.proceed = proceed
12         self.action = action
13         self.action_args = action_args
14         self.action_kwargs = action_kwargs
15
16     def __call__(self, func: Callable[..., Any]):
17         @wraps(func)
18         def wrapper(*args, **kwargs):
```

```
            proceed = self.proceed
            if callable(proceed):
                proceed = proceed(func)
            if isinstance(proceed, bool) and proceed:
                args, kwargs = mutate_params(
                    args, kwargs, self.params_update,
                    ↪ signature(func)
                )
                return func(*args, **kwargs)
            return self.action(*self.action_args,
            ↪ **self.action_kwargs)

        return wrapper
```

5.2 Using the Implemented Advice in an Aspect

The following is an example of an aspect implementation in Python:

```
class Aspect(type):
    # Regular expression used as the pointcut
    before_regexp = re.compile(r"^test[1,2]$")

    def __new__(cls, name, bases, namespace):
        print(f"Modifying the namespace: {namespace}\n")
        for attr_name, attr_value in namespace.items():
            if not callable(attr_value):
                continue

            if attr_name == "__init__":
                # Explicitly calling the decorator instance
                namespace[attr_name] = Before(
                    {"example": "modified_example"},
                    ↪ cls.action, "before __init__", 1, 2
                )(
                    attr_value
                )

            elif cls.before_regexp.match(attr_name):
                # Explicitly calling the decorator instance
                namespace[attr_name] = Before(None, cls.action,
                ↪ f"before {attr_name}", 1, 2)(
                    attr_value
                )
```

```
24
25          return super().__new__(cls, name, bases, namespace)
26
27      @staticmethod
28      def action(arg1: Any, arg2: Any, arg3: Any) -> None:
29          print(f"Doing something {arg1} with args: '{arg2}' and
            ↪  '{arg3}'")
```

`Aspect` is a metaclass that implements the `__new__` method to customize the creation of classes. As you can see, in the `__new__` method, we loop over the namespace of the class, which is being created by the `Aspect` metaclass. Essentially, we loop over the namespace of any class that uses this class as its metaclass. We `continue` the loop if we encounter an attribute that is not `callable`. If the `__init__` method or methods that match the regular expression `before_regexp` are encountered, they are wrapped by chosen advice decorators.

We use regular expressions in concatenation with conditional statements as a way of implementing pointcuts. An alternative to regular expressions is the `fnmatchcase` function of the `fnmatch` Python module. This function allows matching strings with strings that contain Unix-style wildcard characters, although it was originally meant to be used for matching filenames.

If more control over the pointcuts is desired, it is also possible to decorate only methods with a signature having certain parameter types or a certain return value. The signature can be inspected using the `signature` function in the `inspect` Python module. However, such granularity of control over pointcuts requires methods to be type-annotated.

An example use of the `Aspect` metaclass could be the following:

```
1  class MyClass(metaclass=Aspect):
2      def __init__(self, example):
3          self.example = example
4          print(f"INIT: Initializing MyClass with example =
           ↪  '{self.example}'")
5
6      def test1(self):
7          print("TEST 1: Doing something")
8          return 1
9
10     def test2(self):
11         print("TEST 2: Doing something")
12         return 2
13
14     def another_method(self):
```

```
15          print("ANOTHER_METHOD: Doing something")
16          return 3
```

Then we can create an instance of `MyClass` and call its methods:

```
1 my_class = MyClass("example")
2 print(f"Return value: {my_class.test1()}\n")
3 print(f"Return value: {my_class.test2()}\n")
4 print(f"Return value: {my_class.another_method()}")
```

Creating the instance and calling its methods yields the following output:

```
1 Modifying the namespace: {'__module__': '__main__',
  ↪  '__qualname__': 'MyClass', '__init__': <function
  ↪  MyClass.__init__ at 0x0000024C32435620>, 'test1': <function
  ↪  MyClass.test1 at 0x0000024C324356C0>, 'test2': <function
  ↪  MyClass.test2 at 0x0000024C32435760>, 'another_method':
  ↪  <function MyClass.another_method at 0x0000024C32435800>}
2
3 Doing something before __init__ with args: '1' and '2'
4 INIT: Initializing MyClass with example = 'modified_example'
5 Doing something before test1 with args: '1' and '2'
6 TEST 1: Doing something
7 Return value: 1
8
9 Doing something before test2 with args: '1' and '2'
10 TEST 2: Doing something
11 Return value: 2
12
13 ANOTHER_METHOD: Doing something
14 Return value: 3
```

On line 1, you can see that the namespace of `MyClass` contains its attributes, including methods. We skip the decoration of any method not conforming to the `before_regexp` regular expression in the metaclass and therefore the `action` static method of `Aspect` is not executed before it. We can see that the initialization method has also been decorated and the value of the `example` parameter has been modified using the dictionary from the first argument of the `Before` advice.

Note that using metaclasses is a way of automating the process of decorating class methods as opposed to manual decoration using the syntax with the *at* symbol.

5.3 Limitations of AOP in Python

During implementation, we discovered several limitations of using Python within the aspect-oriented paradigm. One of the most limiting factors of Python in this context is its dynamic typing. It is possible that the implementation of a function or method that we want to modify the behavior of will not include type annotations using the `typing` module. In this case, using function signatures for the definition of pointcuts is not a viable option.

Another limitation is the fact that using an aspect-oriented solution like ours in Python relies on direct modification of original source code files, where we need to include the `metaclass` keyword argument in definitions of classes in which we want to modify the methods or add decorators to functions directly. This implies that individual advice or aspect metaclasses that utilize the advice must be imported into the module that houses the functions or classes that we want to modify. However, the implementations of functions, classes, or methods within them are not explicitly modified, implying that any new bugs that might arise are solely within the aspects or advice actions themselves. Assuming the absence of bugs in the original code, fixing or eliminating these aspects or advice actions from the code should resolve the issues encountered.

If a behavior change in class methods is desired, updating class definitions with the `metaclass` keyword argument is necessary for every class intended for this change, even if each class is set to employ the same aspect metaclass. This is a limitation that could lead to code maintainability issues. To overcome this limitation, a different strategy of applying aspects is necessary. A solution to consider in this case could be defining every class with the aspect metaclass and dealing with advice application in the aspect itself based on class names. For such a solution, the use of the `name` argument of the `__new__(cls, name, bases, namespace)` method of the metaclass is required to make decisions whether to apply aspects to the method of a class or not.

Python does not allow multiple metaclasses to be used by a single class. This means that developers cannot use multiple aspects proposed in this work for a single class. Only one aspect metaclass can be used for a chosen class, which could make the aspect metaclass cluttered and difficult to maintain over time. Another challenge may arise when applying AOP in Python to alter the execution of methods within a class that already employs a metaclass. In these scenarios, a workaround involves bypassing the suggested aspects entirely. Instead, we manually apply decorators to the specific methods we intend to modify using the *at* syntax.

6 Evaluation

The solution proposed for AOP in Python was largely inspired by AspectJ and its way of defining aspects and advice within them. It allows us to define advice, such as before, after returning, after throwing, and around, and where to apply them, providing functionality similar to pointcuts in AspectJ.

Similarly to AspectJ, pointcuts can be defined using wildcard characters thanks to the `fnmatch` module in Python. A more flexible approach is using regular expression. However, these could be more error-prone, for example, by defining them incorrectly or making a mistake in their definition, if more complex regular expressions are used.

The implementation allows you to easily change the argument values of methods or functions. It also provides a way of defining which exceptions are to be caught by the after throwing advice. In the around advice, it allows customizing whether to proceed with the advised function or not, similar to AspectJ. Initialization methods are also customizable by applying advice; therefore, AspectJ's ability to define pointcuts for constructor executions is also present in our Python solution.

The implementation conforms to both properties of AOP, quantification and obliviousness [20]. In accordance to the quantification property, it allows performing some actions in programs whenever required conditions arise, e.g. whether to apply advice, whether to proceed with a function or do a different action instead using the around advice, or whether to catch a certain type of exception. Furthermore, the solution can be used to extend existing code with new requirements to the software, in accordance with the obliviousness property.

Considering that AspectC++ implements the same constructs for C++ as AspectJ does for Java, our Python solution also shares similarities with the C++ implementation. However, looking at the aspectlib library written in Python, there are some key differences.

This library does not explicitly provide before, after returning, after throwing, and around advice, but rather provides a decorator named `Aspect`, which can be used to decorate generators. These generators are also decorators that can be used to provide additional functionality to existing functions. The additional functionality is accomplished by using the library's own version of advice - `Return`, `Proceed` or raising an exception by the `raise` keyword. In AspectJ, AspectC++, and our solution, these are not called advice, but rather they are fixtures usable within advice.

Another key difference between aspectlib and our solution is the weaving mechanism. The aspectlib library has a custom weaver function that needs to be run in order to apply aspects to objects. This approach differs from our usage of metaclasses to implement a run-time weaving mechanism.

7 Conclusion

In this paper, we discussed the usage of decorators and metaclasses in Python for aspect-oriented programming. We used them to replicate the constructs that can be found in AspectJ and AspectC++. The chosen constructs were the before, after returning, after throwing and around advice.

These advice were implemented as decorator factories, offering all the fundamental features similar to those found in AspectJ. Metaclasses were proposed as a way to define aspects, similar to aspects in AspectJ. The solution allows for the use of AOP in Python to add functionality to any function or method, including dunder methods, e.g. `__init__`.

The proposed aspect metaclasses implement different advice based on pointcuts. Pointcuts were defined using regular expressions or wildcard characters. Furthermore, pointcuts can be extended using the `name` argument of the `__new__` method in metaclasses.

The limitations of aspect-oriented programming in Python were evaluated in comparison with the state-of-the-art language, AspectJ. Alternative approaches to using our solution were also considered as ways to overcome limitations. The proposed solution lacks the more advanced features present in AspectJ; however, the implemented advice could be useful for extending existing code with new requirements to the software.

Acronyms

AOP aspect-oriented programming

References

[1] Heba A Kurdi. "Review on Aspect Oriented Programming". In: *International Journal of Advanced Computer Science and Applications*. Vol. 4. 9. The Science and Information Organization. 2013, pp. 22–27. URL: https://thesai.org/Downloads/IJACSA_Volume4No9.pdf.

[2] Massood Khaari and Raman Ramsin. "Process Patterns for Aspect-Oriented Software Development". In: *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*. IEEE. 2010, pp. 241–250. DOI: 10.1109/ECBS.2010.33.

[3] Arne Bachmann, Henning Bergmeyer, and Andreas Schreiber. "Evaluation of Aspect-Oriented Frameworks in Python for Extending a Project With Provenance Documentation Features". In: *The Python Papers* 6.3 (2011), p. 3. URL: https://core.ac.uk/download/pdf/230920601.pdf.

[4] Luciano Ramalho. *Fluent Python 2nd Edition*. O'Reilly Media, Inc., 2022. ISBN: 978-1-492-05635-5.

[5] Anand Rathore. *MetaClasses in Python*. May 2023. URL: https://codesarray.com/view/MetaClasses-in-Python (visited on 10/16/2023).

[6] Ramnivas Laddad. *AspectJ in Action Second Edition*. Manning Publications Co., 2010. ISBN: 978-1-933988-05-4.

[7] TJ O'Connor. *Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers*. Syngress, 2013. ISBN: 978-1-59749-957-6.

[8] Gregor Kiczales et al. "Aspect-oriented Programming". In: *ECOOP'97 Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*. Springer. 1997, pp. 220–242. DOI: 10.1007/BFb0053381.

[9] Gregor Kiczales et al. "An overview of AspectJ". In: *ECOOP 2001 Object-Oriented Programming: 15th European Conference Budapest, Hungary, June 18–22, 2001 Proceedings 15*. Springer. 2001, pp. 327–354. DOI: 10.1007/3-540-45337-7_18.

[10] Andy Clement et al. *Eclipse AspectJ: Aspect Oriented Programming With AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley Professional, 2005. ISBN: 0-32-124587-3.

[11] The AspectJ Team. *The AspectJ Programming Guide*. Palo Alto Research Center, Incorporated, 2003. URL: https://eclipse.dev/aspectj/doc/released/progguide/index.html (visited on 10/30/2023).

[12] Joseph D Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-oriented Programming in Java*. John Wiley & Sons, 2003. ISBN: 0-471-43104-4.

[13] Stack Overflow. *2023 Developer Survey*. 2023. URL: https://survey.stackoverflow.co/2023 (visited on 10/31/2023).

[14] Erich Gamma et al. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co., 1995. ISBN: 0201633612.

[15] Vaskaran Sarcar. *Java Design Patterns: A Hands-On Experience with Real-World Examples, Third Edition*. Apress Berkeley, CA, May 2022. ISBN: 978-1-4842-7971-7.

[16] Guido Van Rossum and the Python Development Team. *The Python Language Reference*. 2023. URL: https://docs.python.org/3.12/reference/ (visited on 10/31/2023).

[17] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language". In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. 2002, pp. 53–60.

[18] Ionel Cristian Mărieș. *Aspectlib Documentation*. Nov. 2018. URL: https://python-aspectlib.readthedocs.io/ (visited on 12/04/2023).

[19] Martin Matusiak. *Strategies for aspect oriented programming in Python*. May 2009. URL: https://matusiak.eu/media/uploads/aop_strategies.pdf.

[20] Robert E Filman, Daniel P Friedman, and Peter Norvig. "Aspect-Oriented Programming is Quantification and Obliviousness". In: *Workshop on Advanced Separation of Concerns*. RIACS, 2000. URL: https://homepages.cwi.nl/~storm/teaching/reader/FilmanFriedman00.pdf.