Strategies for aspect oriented programming in Python

Martin Matusiak

May 18, 2009

Contents

1 Introduction

The paradigm of Aspect Oriented Programming (AOP) has emerged in recent years as a potential measure to alleviate the complexity of software design and maintenance. AOP is readily applicable to object oriented software, although adoption has been sporadic to date. The best known implementation is AspectJ[2] (for the Java ecosystem), which can be considered a reference implementation for aspect oriented programming.

AOP is essentially a disciplined, methodical way of injecting code into existing source files. It is defined by AspectJ as an out-of-band code transformation with a separate compiler, taking the original source files, plus the so called "aspect code" (which defines these injections, and is still Java code) to produce standard Java class files. The advantage is clear: the original source is untouched, and aspect code can add new behaviors which can be turned on or off just by moving between the Java compiler and the AspectJ compiler. This has obvious advantages as a step between development/testing and production.

AOP defines a nomenclature for code injection.

1. Code injection is possible at certain well defined points, such as before a method call, after a method body etc. These points are called *join points*.
2. Code injection is declared in units called *advice*, which provide the code to be injected.
3. Advice are matched against join points using string matching (globbing or regular expressions). Every advice has a matching string, and it is called a *pointcut*.
4. Advice are collected in classes (or other executable units) called *aspects*.

1.1 AOP in Python

Unlike AspectJ in Java, Python does not have a canonical AOP framework. It can be debated to what extent aspect oriented practices are useful or applicable to Python's dynamic nature. Nevertheless, there have been a number of projects to enable aspect oriented programming in Python.

The following framworks are covered in this paper.

1. aopy[1]
2. Aspyct[3]
3. Lightweight Python AOP[4]
4. Logilab aspects[5]
5. PEAK[6]
6. Pythius[7]
7. Spring Python[8]

The following framworks are not covered.

1. PyPy AOP[9]

Aspect oriented programming in PyPy is an AOP extension for the PyPy compiler environment and is thus not on equal footing with the other implementations, all of which require only CPython.

1.2 Scope

It is my mission in this paper to examine a number of different strategies for AOP in Python, and relate how different implementations have leveraged these.

It is the focus of this paper to explore the *strategies* behind code injection applied to aspect oriented programming. The paper does not serve as a general review of the various frameworks in order to provide a scoresheet. Instead, it is my view that the injection (or hook) mechanism itself – that which makes code injection possible – is the crucial and defining property of any aspect oriented approach, and is therefore my focus. Outside of this I make no attempt to assess the merits of the implementations presented.

2 Useful language facilities

Python has a couple of language features that promote aspect oriented programming in that they provide access to the underlying data structure without disrupting the interface. In other words, they are means of instrumentation that are invisible to clients of the code. The various AOP frameworks use these features widely.

2.1 Properties

Properties enable an indirection on access to variables of class instances.[1] The following example serves to illustrate.

```python
class Class(object):
    def __init__(self):
        self.att = 1

    def _get_(self):
        print "get", self._att
        return self._att

    def _set_(self, value):
        print "set", value
        self._att = value

    def _del_(self):
        print "del"

    att = property(fget=_get_, fset=_set_, fdel=_del_)

cls = Class()
#> set 1
print cls.att
#> get 1
#> 1
cls.att = 2
#> set 2
print cls.att
#> get 2
#> 2
del cls.att
#> del
```

[1]Properties only apply to instance variables, class variables cannot be wrapped in properties. Properties are also limited to new style classes.

Lines 1-3 show a standard class definition, with a variable `att` initialized with a value. Ordinarily, access to this variable will access the object directly. However, in this case we wish to introduce an indirection on this attribute.

Lines 5-14 define three functions that will receive calls upon access to `att`. Line 16 is the crucial statement, which reassigns the variable `att` to an instance of a property given the three functions defined above. The property mechanism will now dispatch all attribute access on `att` to `_get_`, it will dispatch assignment on `att` to `_set_`, and it will call `_del_` just before `att` is garbage collected. The functions `_get_` and `_set_` actually operate on an attribute called `_att`, which is a variable known only to these two functions, and serves to store the value of the attribute.[2]

Line 18 instantiates the class `Class` with an instance `cls`. This executes the `__init__` method, where `att` is assigned a value. Since `att` is wrapped in a property, this assignment is routed through the method `_set_`, which performs the actual assignment. Subsequent access and assignment to `att` passes through `_get_` and `_set_` respectively, before the attribute is deleted, which triggers `_del_`.

Properties provide a practical solution to the problem of imposing logic on access or assignment to selected attributes, but without changing the interface. Thus access to a property does not differ from access to any other object from the client's point of view.

2.2 Decorators

A decorator wraps a function much in the same way that a property wraps an attribute. A decorator is a function which accepts the function it decorates, and returns a replacement function. The following example demonstrates use of a decorator.

```
1  def decorator(func):
2      def new_func(x):
3          print "Received input value: %s" % x
4          res = func(x)
5          print "Returning output value: %s" % res
6          return res
7      return new_func
8
9  @decorator
10 def func(x):
11     return x+1
12
13 print func(1)
14 #> Received input value:  1
15 #> Returning output value:  2
16 #> 2
```

Lines 10-11 show a function definition. Line 9 shows the special syntax for applying a decorator to a function. The decorator itself is defined on lines 1-7. The outer function `decorator` accepts a function `func`, defines in its body a function `new_func`, and returns `new_func`. The net effect is that `new_func` replaces `func` in the module.

However, `new_func` is defined in the scope of `decorator`, and is actually a closure (ie, it has acess to the scope of `decorator`). And so in the body of `new_func` there is a call to the original function `func`. The result of that call is captured in `res`, and returned by `new_func`. The parameter `x` was received by `new_func`, and is forwarded to `func`. But before and after this call other code can be executed.[3]

[2]This is purely as a matter of convention; nothing prevents other functions from access to `_att`. However, a name different from `att` is necessary, otherwise access to `att` in the body of `_get_` will trigger a recursive call to `_get_`, causing infinite recursion.

[3]While it is not shown here, the Python interpreter provides further introspection into the arguments to `func`, their names and values, and these values can be mutated in useful ways. For instance, it is possible to determine if the first argument is called `self` and whether it is an instance of the class the function belongs to.

The call to `func` can even be scrapped altogether.

2.3 Metaclasses

Formally, a metaclass relates to a class in the way that a class relates to an instance. The *type* (or class) of an instance is its class; the *type* (or class) of a class is its metaclass. Python has a standard metaclass used for all classes which do not define their own, called `type`.

Just as properties and decorators can be used to instrument attributes and functions one by one respectively, metaclasses provide access to the whole namespace of a class. And thus metaclasses can be used to set properties and decorators in bulk, programmatically. They can also mutate a class in any other way. The following snippet exemplifies one use of a metaclass.

```
 1  class Printable(type):
 2      def __new__(cls, name, bases, dct):
 3          dct['__str__'] = lambda self:\
 4                  "I'm an instance of %s" % self.__class__.__name__
 5          return type.__new__(cls, name, bases, dct)
 6
 7  class Class(object):
 8      __metaclass__ = Printable
 9
10  print str(Class())
11  #> I'm an instance of Class
```

Lines 7-8 define a class `Class`, without methods. A `__metaclass__` variable is set at class level (a class attribute).

Lines 1-5 define a metaclass. The metaclass derives from `type`, not from `object` as classes do. And it defines a meta-method `__new__`, which accepts the name of the class to be instantiated (in this case `Class`), the bases classes of the class (`object`), and the members of the class (the attributes and functions).

Line 3 sets the attribute `__str__` in the dictionary (namespace) of the class, which serves to override the standard method `__str__` with a custom version. `__new__` then delegates to `type.__new__`, which creates the class and returns it.

Line 10 prints the string representation of an instance of `Class`, which corresponds to the override on line 3.

Python 3.0 will also have class decorators, an alternate form of mutating classes which is analogous to function decorators.[10]

3 Strategies

3.1 In-source modifications

The bulk of the AOP frameworks for Python rely on mutating the namespace from within the module itself. The demonstration snippets they provide typically show the aspects being applied in the same module as the code to be instrumented. Figure 1 illustrates this pattern.

The objects that are to be instrumented have to be in scope by the time the instrumentation statements are executed. Use of the objects must then follow the instrumentation statements. Additional modules can be instrumented, either in-source, or in the main module, once they have have been imported into the main module's scope. Dynamic mutation is informally known as *monkey patching*.[11]

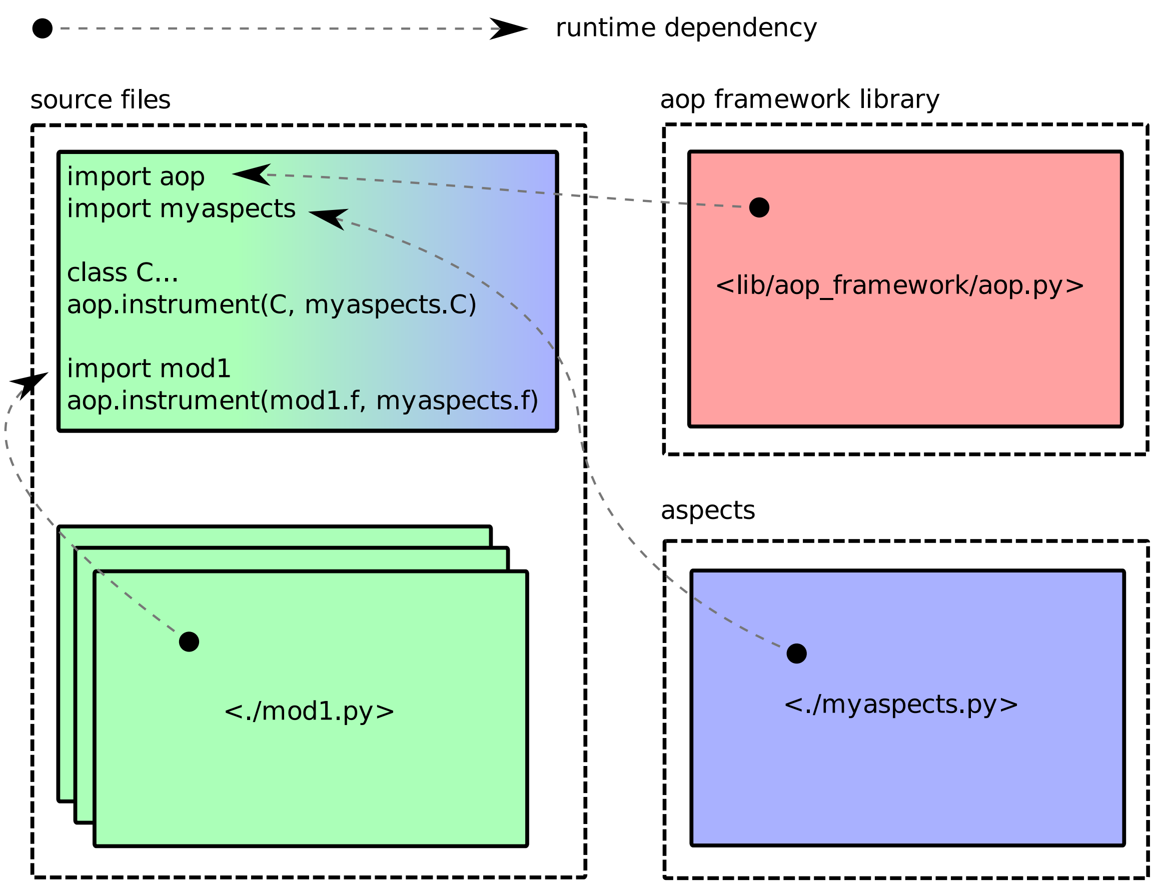

Figure 1: In-source pattern

3.1.1 Metaclass as hook mechanism

When an object is instantiated from a class, the `__init__` method is called on the instance. Just the same way, a class is instantiated from a metaclass, and the `__new__` method is called as a constructor for the class. This opens a window of time in which the class can be mutated before any clients will have had the chance to use it. This can be seen as a static mutation, given that there is no case in which a client may have accessed the class or its instances before the mutation took place.

Python objects deriving from `object` inherit the metaclass `type`. A class can override this by setting the `__metaclass__` attribute at class level. This can only guarantee a static mutation if the metaclass is set in the body of the class declaration, so that it takes effect when the class is compiled into a code object. Otherwise the class may already have been accessed by clients, which would have seen the pre-mutation state.

Pythius is perhaps the most referenced Python AOP framework (although its purpose is broader it does include an AOP component) and uses the metaclass as a hook into the class. The following example is from Pythius's demonstration code.

```
 1  import aop
 2
 3  class OffByOne(aop.Aspect):
 4      ...
 5
 6  obo = OffByOne()
 7
 8
 9  class Student(object):
10      __metaclass__ = aop.Metaclass
11      _aspect = obo
12      ...
13
14  student = Student()
```

Line 3 begins a definition of an aspect. The aspect (in the form of a regular class) is then instantiated on line 6. This can be readily moved to a dedicated “myaspects” module if so desired. Then follows the definition of a class on line 9. This class sets a metaclass, pointing to the metaclass defined by the AOP framework itself. Then the attribute `_aspect` is set to the instance of the user defined aspect. Finally, the class is instantiated.

All of the logic external to the operation of the user defined code (ie. the module minus all AOP related code) can be moved to a separate “aspect“ module, except for the indispensable metaclass declaration on line 10. (A user defined metaclass could just as well handle all the AOP logic, the `_aspect` attribute is not essential.)

Consequently, this technique does require setting the metaclass for all classes that are to be instrumented, but is not very invasive. However, it is not a general technique for AOP since it only applies to classes. Functions and attributes at module level do not have an equivalent hook, so they may not be instrumented.

3.1.2 Dynamic mutation

The obvious alternative is to intermingle the main program code with aspect code, mutating the namespace without trying to keep the aspect code external (not necessarily the declaration of the aspect code, but certainly its application onto the program code). A number of Python AOP frameworks take this approach, including Aspyct, Lightweight Python AOP, Logilab aspects and PEAK.

The following example is from Logilab’s tutorial code and is demonstrative of how all of these frameworks work. They vary in the richness of services offered, but the invocation mechanism is the same – binding in the current namespace.

```
 1 from logilab.aspects.weaver import weaver
 2 from logilab.aspects.lib.logger import LoggerApsect
 3 import sys
 4
 5 stack = StackImpl()
 6
 7 # Push an element on the stack, the method call is not traced
 8 stack.push("an element")
 9
10 # Weave aspect code (log will be done on sys.stderr)
11 weaver.weave_methods(stack, LoggerAspect, sys.stderr)
12
13 # Push an other element, method call will now be traced
14 stack.push("another element")
15
16 # Unweave logger aspect
17 weaver.unweave(stack, LoggerAspect)
18
19 # Now, call methods aren't traced anymore
20 stack.push("a third element")
```

Lines 1 and 2 load the AOP framework module, as well as one of the pre packaged aspect modules. Line 5 instantiates an object from a user defined class, followed by line 8, which calls a method on it. Next comes the application of the aspect in scope onto the user object. The next method call on line 14 happens on a mutated version of the `push` method, with a logging side effect. Finally, the aspect can be detached from the object again.

Dynamic mutation has no restrictions on access to the code module, so it obviously has all the power of any other Python code. All objects can be rebound and mutated at any time, which makes it possible to enable and disable aspects dynamically. Unfortunately, this also means that static mutation, should it be desired, becomes a matter of code style, as it is not enforced by the language in any way. This could be a source of bugs if clients erroneously access objects before they have been mutated and experience disparate results (especially with respect to side effects).

Dynamic mutation is also highly invasive in how it mingles the code in the module with aspect code.

3.2 External invocation

External invocation is an orthogonal approach to in-source modification. If modifying the source is deemed undesirable, then code injection can still be achieved by invoking the program in a non-standard way, eg. for the purpose of debugging or benchmarking.

Figure 2 illustrates this method. The source is untouched. Instead, the module in question is imported and executed in an external module, where it can be instrumented.

3.2.1 External metaclass override

Metaclasses are declared either in the body of a class definition, a base class, or the global `__metaclass__` variable.[4] This makes it possible to set a global metaclass for all the classes defined in a module (provided they do not define a metaclass), which can mutate the classes accordingly. As mentionded in section 3.1.1, mutation by metaclass is a static mutation.

The following example illustrates this method.

[4]For precise resolution order see [12].

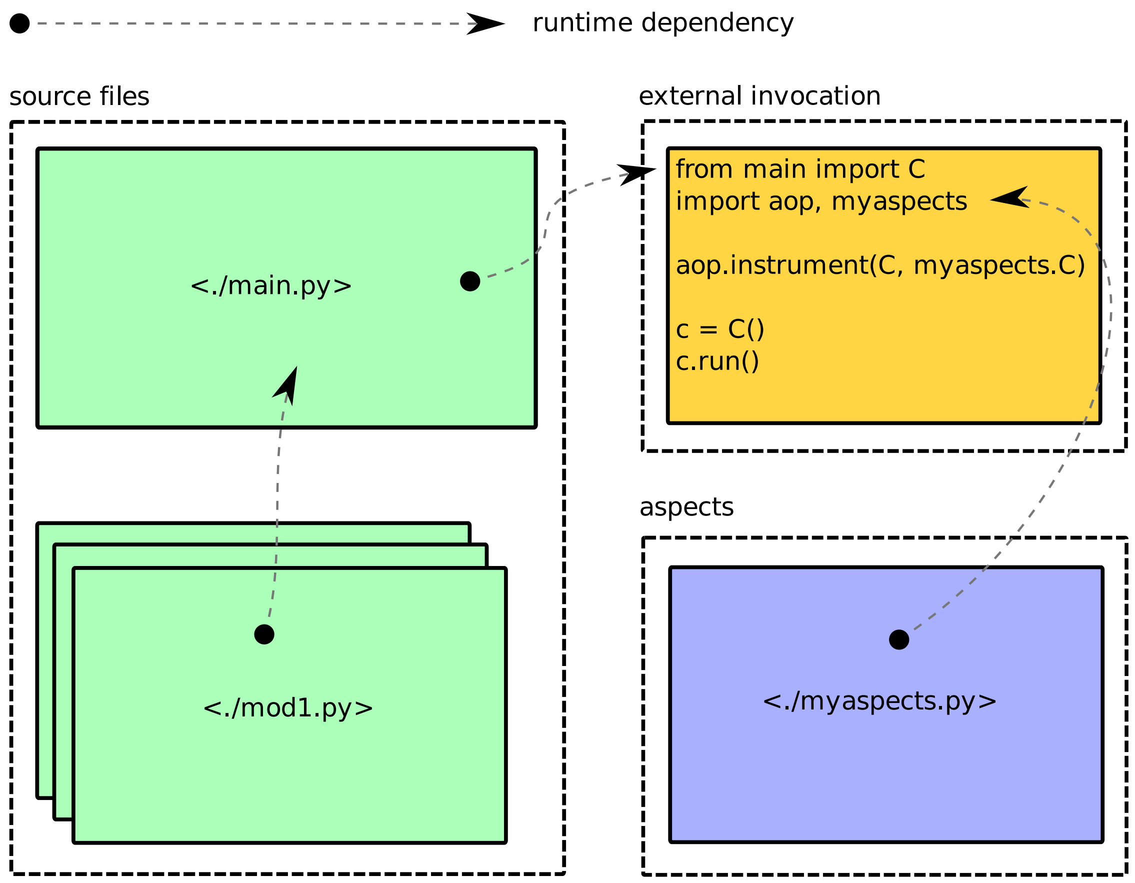

Figure 2: External invocation pattern

```
 1  # <./main.py>
 2  class Number():
 3      def display(self):
 4          print 45
 5
 6  if __name__ == "__main__":
 7      Number().display()
 8
 9
10  # <./external.py>
11  class Verbose(type):
12      def __new__(cls, name, bases, dct):
13          print "Creating class", name
14          return type.__new__(cls, name, bases, dct)
15      def __init__(cls, name, bases, dct):
16          print "Initializing class", name
17          super(Verbose, cls).__init__(name, bases, dct)
18
19  env = globals().copy()
20  env['__metaclass__'] = Verbose
21  execfile('main.py', env, {})
22  #> Creating class Number
23  #> Initializing class Number
24  #> 45
```

Lines 2-4 define the class `Number` with a method `display`. Line 6 carries the idiomatic stanza for modules that are executable, stating that the following code will execute if this module is the current main module. Line 7 instantiates an object of class `Number` and calls the method `display`.

Lines 11-17 define the metaclass `Verbose`. The `__new__` method prints output just before creating the class, then calling `__new__` on its base-metaclass `type` (line 14). The `__init__` method is then called to initialize the class[5], which again delegates to the base of `Verbose`.

On line 19 the global namespace is cloned (so that it can be mutated) and the `__metaclass__` attribute is set. Finally, the target module `main.py` is executed given the global namespace defined in `env` (note that this dictionary also contains the attribute `__name__`, derived from the `external.py` module, which will trigger the stanza on line 6), and an empty local namespace. What follows is output from `main.py`: first the class `Number` is instantiated, with output from `__new__` and `__init__`, then comes the output from the method `display`.

This method is not invasive, unlike the similar metaclass hook from secion 3.1.1, as it is external to the module. However, it only applies to old style classes, which are deprecated.[6] New style classes, in the absence of a `__metaclass__` attribute in the class body, will inherit the metaclass `type`.

Furthermore, this approach only allows instrumenting the module being executed. Any classes defined in modules imported by `main.py` will be compiled by the time they appear in `main`'s namespace, and unaffected by main's `__metaclass__` attribute. Other clients of the `main` module will not be affected by this at all.

3.2.2 External dynamic mutation

Dynamic mutation in-source (3.1.2) has an external counterpart. The strategy is to import the modules of choice and mutate the objects before use.

The following is an example.

```
 1  # <./main.py>
 2  def func():
 3      return 1
 4
 5
 6  # <./external.py>
 7  def dec(f):
 8      def new_f(*a, **k):
 9          print "---- dec ----", f.__name__
10          return f(*a, **k)
11      return new_f
12
13  import main
14  main.func = dec(main.func)
15  print main.func()
16  #> ---- dec ---- func
17  #> 1
```

The `main` module defines a function `func`. This is a function we wish to instrument. Lines 7-11 in the `external` module define a simple decorator that wraps the function it is applied to. On line 14 the name `main.func` is rebound with the decorator `dec`. Then the function is called.

The `func` object is only mutated in the `external` module, and other clients will see the unmodified object, in contrast to the in-source variant of this method.

[5]Note that this is not the same `__init__(self)` method called on instances of a class, this `__init__` initializes the class, not instances. The parameter list is evidence of this.

[6]Old style classes will not appear in Python 3.0, see [13].

3.2.3 Proxy objects

Proxy objects are widely used in technologies that handle remote procedure calls. The design pattern can also be used to introduce indirection on local objects for the purpose of aspect oriented programming. Spring Python is a framework that applies external invocation through proxy objects.

The following is an abridged example from Spring Python's demonstration code.

```
 1  # <./module.py>
 2  class SampleService:
 3      def method(self, data):
 4          return "You sent me '%s'" % data
 5
 6
 7  # <./external.py>
 8  from springpython.aop import MethodInterceptor, ProxyFactor
 9  from module import SampleService
10
11  class WrappingInterceptor(MethodInterceptor):
12      def invoke(self, invocation):
13          results = "<Wrapped>" + invocation.proceed() + "</Wrapped>"
14          return results
15
16  factory = ProxyFactory()
17  factory.target = SampleService()
18  factory.addInterceptor(WrappingInterceptor())
19  service = factory.getProxy()
20
21  print service.method("something")
22  #> "<Wrapped>You sent me 'something'</Wrapped>"
```

Lines 1-4 show the program source, which defines a class `SampleService`. The `external.py` module imports the main module and objects from the AOP framework. Line 11 defines the interceptor `WrappingInterceptor`, which will be applied to the class `SampleService`. Lines 16-19 instantiate a proxy for an instance of `SampleService` and apply the `WrappingInterceptor` on all methods of the class. Eventually, the proxy object is bound to the name `service` and can be used just the same as instances of the unmodified `SampleService` class.

According to the rationale of Spring Python, the method of proxy objects is intended to be used at junction points between modules of a program (third party libraries, for instance), without the need to touch the program source. This can indeed be realized to provide many connection-point types of servies, like a security layer, as a container that catches all exceptions, and so on. But this would make the framework more of a middleware than an aspect oriented implementation. After all, it is the stated goal of AOP to be able to instrument code at any depth, not merely at the interface of a component.

3.3 Program transformation

The strategies presented thus far require either changes to the code, or a custom execution model. They are not out-of-band methods of instrumentation – they do not enable running the program completely unmodified. aopy is a proof of concept AOP implemention I wrote to demonstrate an out-of-band instrumentation technique whose effect is similar to that of AspectJ: the program is transformed in the compilation phase and subsequently executed as normal.

The pattern is laid out in figure 3. The source files are compiled to bytecode, through a process of program transformation. This transformation is defined by the specification (spec) file, which pairs units of code injection (properties, decorators, metaclasses) with pointcuts (patterns to match against

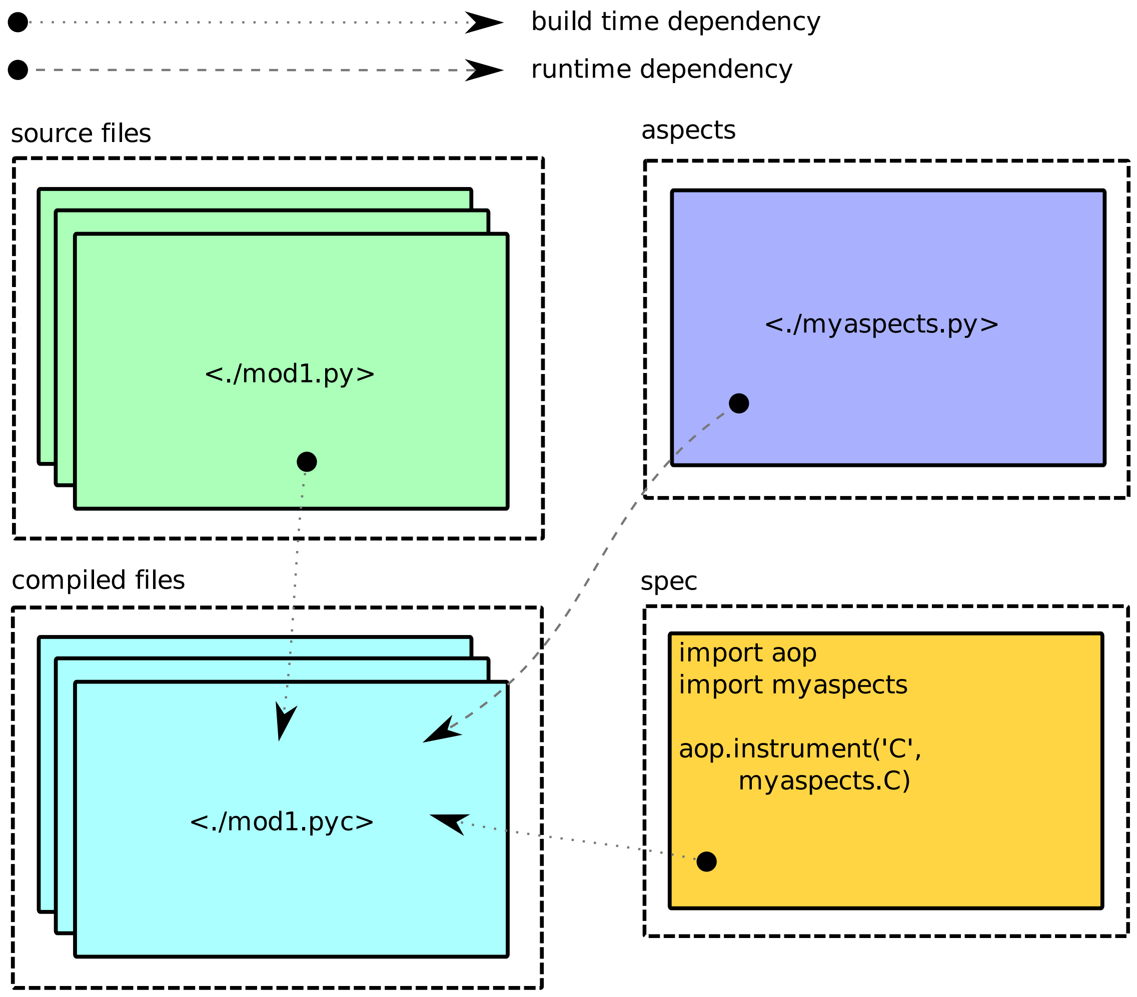

Figure 3: Program transformation pattern

code entities in the source files). The compilation produces instrumented bytecode with references to the aspects.

What follows is a demonstration of aopy use.

The program source

```
 1 # <./main.py>
 2 class Class(object):
 3     def __init__(self):
 4         self.att = 1
 5
 6     def compute(self, x):
 7         return x**x
 8
 9 cls = Class()
10 print cls.att
11 #> 1
12 print cls.compute(4)
13 #> 256
14 print str(cls)
15 #> <__main__.Class object at 0x7f61fd8a2210>
```

The code to be instrumented is held in the module `main`. Lines 2-7 define a simple class with one attribute `att` and one method `compute`. The remainder demonstrates first instantiating the class, then accessing the attribute, then calling the method, and finally showing the string representation of the instance. This output will change as the class is instrumented.

A transformation

The module `myaspects` contains the code elements that will be injected into `main`.

```
1  # <./myaspects.py>
2  def set_att(self, value):
3      print "set", value
4      self._att = value
5
6  def get_att(self):
7      print "get", self._att
8      return self._att
9
10
11 def decorator(func):
12     def new_func(*args, **kwargs):
13         print "Received arguments:", args, kwargs
14         res = func(*args, **kwargs)
15         print "Returning result:", res
16         return res
17     return new_func
18
19
20 class Metaclass(type):
21     def __new__(cls, name, bases, dct):
22         def __str__(self):
23             members = ", ".join([m for m in dir(self)])
24             classname = self.__class__.__name__
25             return "%s instance: %s" % (name, members)
26         dct['__str__'] = __str__
27         return type.__new__(cls, name, bases, dct)
```

Lines 2-8 define two functions `set_att` and `get_att`. These will indeed become methods in the class `main.Class`, and accordingly take `self` as first parameter. Their purpose is to administer the attribute `_att`, which will become the internal name for the attribute `att` in the class `main.Class`. `att` which will become a property.

Lines 11-17 define a decorator function `decorator`, which will wrap the method `compute` in `main.Class`. This decorator defines a replacement function `new_func`, which displays the arguments received before calling the function it wraps, then makes the call, and then displays the result before returning it.

Lines 20-27 define a metaclass `Metaclass`, which will become the metaclass of `main.Class`. The constructor function `__new__` injects a replacement for the standard `__str__` method, listing all the members of the class.

```
1  # <./spec.py>
2  import aopy
3  from myaspects import *
4
5  aspect = aopy.Aspect()
6  aspect.add_property('main:Class/att', fget=get_att, fset=set_att)
7  aspect.add_decorator('main:Class/compute', decorator)
8  aspect.add_metaclass('main:Class', Metaclass)
9
10 __all__ = ['aspect']
```

The module `spec` defines the code injection that will be performed on `main` using the items in `myaspects`.

The `Aspect` class is the public API offered by aopy. Line 5 creates a new `Aspect` instance, which represents a semantic unit of code injection.

The next step is to populate the aspect with code instrumentation items. Every item consists of two parts, a pointcut (a matching pattern), and an advice (a unit of code injection). Pointcuts are defined as *pathspecs* (pairs of paths); the first part is the path of the module,[7] the second is the path of the element in the module. Pathspecs are regular expressions.

Line 6 adds a property to the aspect. This means that an attribute, if matched by the pointcut, will be wrapped as a property. The pathspec given will match an attribute `att` in a class `Class`, in a module `main`.

Line 7 adds a decorator to the aspect. This means that any function (or method) matching the pointcut will be decorated with this decorator. The pathspec given will match a function `compute` in a class `Class`, in a module `main`.

Line 8 adds a metaclass. The pathspec will match any class named `Class` at the top level of a module named `main`.

Line 10 sets the standard Python module attribute `__all__`, which lists the aspect objects to be exported by this spec.

The transformed program

The following listing displays the transformed program. This file is not actually produced by aopy – the transformed abstract syntax tree is compiled directly to a bytecode module. But its content correponds to the code shown.

```
 1  # <./main.py> transformed
 2  import myaspects+
 3
 4  class Class(object):
 5      __metaclass__ = myaspects.Metaclass+
 6
 7      def __init__(self):
 8          self.att = 1
 9
10      @myaspects.decorator+
11      def compute(self, x):
12          return x**x
13
14      att = property(fget=myaspects.get_att, fset=myaspects.set_att)+
15
16  cls = Class()
17  #> set 1+
18  print cls.att
19  #> get 1+
20  #> 1
21  print cls.compute(4)
22  #> Received arguments:  (<__main__.Class object at 0x7fe7c4274750>, 4) {}+
23  #> Returning result:  256+
24  #> 256
25  print str(cls)
26  #> Class instance:  __class__, __delattr__, __dict__, __doc__, ...*
```

Lines that differ from the original program source have been marked. A red plus + marks a new line, a blue dot * marks a changed line.

The first (executable) line in the program, line 2, contains a new import statement, which brings the objects that have been injected into scope. The injected import statement will always precede any other imports.

[7]The module declared in a pathspec will be matched by name, it does not refer to any particular file, merely the path of a module in a package hierarchy. If the pattern matches several modules with name `main`, all will be successful matches.

The next new line is line 5, which is a metaclass declaration, referencing `myaspects.Metaclass`. Any existing metaclass declaration will have been replaced by this one.

Line 10 shows the decorator `myaspects.decorator` applied to the method `compute`. The injected decorator will be applied after any existing decorators (an outer wrap).

Line 14 assigns a property on the attribute `att`, with the functions `get_att` and `set_att` from `myaspects`. These functions will be called from within the class and behave just like any other methods in the class. Any existing property on `att` will be replaced with this statement.

The remainder shows use of the class in an instrumented state.

First, we see the property in action. Line 17 now appears, because the `set_att` function is called from the `__init__` method when the class is instantiated. Line 19 appears because the `set_att` method is called when the attribute is accessed.

Line 21 invokes the decorator, which wraps `compute`. This produces lines 22 and 23. `compute` is called with two arguments, the first is an instance of the class (the `self` argument), the second is the integer bound to `x` in the function. There are no keyword arguments, so the dictionary `kwargs` is empty as shown.

Finally, line 26 is changed because the call to `__str__` on the instance of `Class` is made to a custom method `__str__`, set by the metaclass of `Class`.

The program transformation strategy has the marked advantage of being an out-of-band instrumentation method. Consequently, all mutations are static and thus the lifetime of an object will never affect its instrumentation state.

The disadvantage of this method is that the injected code is invisible to the programmer, and thus more difficult to debug. Both the aspect modules and the spec modules are regular Python code, which can be run standalone, and aspects can be applied in-source for testing. But the final instrumented target code is not available.

aopy is written using the `compiler` module of the standard library, which replicates the C compiler entirely in Python. This module is deprecated and will not be available in Python 3.0, see [14]. The Python standard library retains functionality to parse modules into abstract syntax trees and perform transformations on them through a new `ast` module. However, no compilation mechanism from AST to bytecode has yet been announced (presumably a function to compile AST using the C compiler will become available to complete the chain).

4 Evaluation

The task of evaluating the various strategies must start from a set of values which are thought as important criteria for what a given implementation has to offer. The origin of aspect oriented programming is said to be Gregor Kiczales's paper from 1997 ([15]). The realization of this concept is indeed AspectJ, which originates from the same research group. However, in the decade that followed, dozens of implementations ([16]) have sprung up (I present those related to Python in this paper) and it may be the case that AOP has evolved somewhat from the original concept. Notwithstanding this possibility, I take the view that the purpose of AOP remains to be the practice of instrumentation through code injection.

The following subsections offer an evaluation of the various strategies on criteria of interest. A summary is shown in the matrix below. I stress that this is not an evaluation of the AOP frameworks themselves, it is an assessment of the strategies presented, and the potential they have. This does not imply this potential has been realized in all, or even any, of the implementations.

Strategy comparison matrix					
Strategy	**Depth**	**Reach**	**Invasive**	**Lifecycle**	**Out-of-band**
Metaclass hook	Class	Full	Minimal	Static	No
Dynamic mutation	Stmt wo/side eff.	Full	Yes	Dynamic	No
External metaclass	Class	Partial	No	Static	No
External dynamic mut.	Stmt wo/side eff.	Partial	No	Dynamic	No
Proxy objects	Stmt wo/side eff.	Partial	No	Dynamic	No
Program transformation	Stmt w/side eff.	Full	No	Static	Yes

Figure 4: Strategy comparison matrix

4.1 Injection depth

All strategies that either mutate the namespace from within the module itself, or which rely on mediating access to the module, have access to the module's full namespace and can rebind symbols at will (including bindings in imported modules). This effectively enables injection down to the statement level, excluding side effects.[8] The strategies in this category are dynamic mutation (both in-source and external) and proxy objects.

Program transformation has access to the full syntax tree and can can thus inject at the statement level as well, but can also preempt side effects by mutating (or branching off from) statements before they are executed.

Metaclass based strategies are more limited, in that they can only mutate members of a class (methods and variables). Functions not contained in classes, as well statements at module level, cannot be effected.

4.2 Reach

In-source modifications and program transformation are strategies that have full reach, because the mutations happen in the module itself, so any client importing the module will see the mutated state of the objects.

External invocation strategies, conversely, must insist that all clients access the module through an intermediary, so they only have partial reach.

4.2.1 A counter example to full reach

Suppose there is a module that contains a function `f`. This is a function we wish to rebind for clients of the code.

```python
#<./module.py>

def f():
    print("original function")
```

To that end, we create a proxy module that imports the code module and rebinds the function.

[8]Side effects here means calls to other modules or side effects unto the operating system at large. A statement which deletes a file cannot be reversed by rebinding symbols post-execution.

```
#<./proxy.py>

import module

def new_f():
    # module.f() # dangerous code, replace with:
    print("instrumented function")

module.f = new_f
```

Clients of the proxy will now see the rebound version of the function:

```
#<./proxyclient.py>

from proxy import *

module.f()

#> instrumented function
```

But what if a client imports the module directly? The function appears in its original incarnation, and we have failed to intercept this function call through our instrumented function.

```
#<./moduleclient.py>

import module

module.f()

#> original function
```

4.3 Invasiveness

Strategies that employ external invocation are uninvasive, because all the instrumentation is external to the source module. This also applies to program transformation

The metaclass hook method is invasive, but in a minimalistic way. The metaclass has to be set in each class, but the rest of the aspect code can be external.

Dynamic mutation mixes source code with instrumentation in the same module and is therefore in my view quite contrary to the objective of AOP: separating source code and aspects. The two parts can be kept separate in the module by convention, first listing all the objects, and then the instrumentation declarations, but that is still only a matter of convention and therefore not a firm level of separation.

4.4 Instrumentation lifecycle

Metaclass based strategies are static mutations, which guarantee that the object will have been mutated by the time it comes into scope and clients can access it. Program transformation is also a static mutation, because the module is compiled to bytecode in mutated form before it is ever executed.

Proxy objects is a dynamic mutation method that is intended to be used statically. Since it does not have full reach, it becomes a matter of convention to enforce that objects from a module are always accessed through proxy objects. Proxy objects could also be used dynamically to enable/disable instrumentation.

Dynamic mutation strategies are, as the name suggests, dynamic. As with proxy objects, they can be used statically by convention, but their applicability is perhaps more valuable as a genuinely dynamic method of enabling instrumentation in response to stimuli in the course of execution (for instance to attach a logger to a misbehaving component just long enough to diagnose a problem).

4.5 Out-of-band

Program transformation is the only strategy that allows out-of-band instrumentation, that is code injection without either changes to the source files or a custom invocation model.

References

[1] Martin Matusiak. *aopy: A program transformation-based aspect oriented framework for Python* 2009.

[2] The AspectJ Project *http://www.eclipse.org/aspectj/* 2008.

[3] Aspyct *http://aspyct.sourceforge.net/* 2008.

[4] Lightweight Python AOP *http://www.cs.tut.fi/~ask/aspects/* 2008.

[5] Logilab aspects *http://www.logilab.org/projects/aspects* 2008.

[6] Python Enterprise Application Kit *http://peak.telecommunity.com/* 2007.

[7] Pythius *http://pythius.sourceforge.net*/ 2002.

[8] Spring Python Aspect Oriented Programming *http://springpython.webfactional.com/wiki/AspectOrientedProgramming* 2008.

[9] PyPy Aspect Oriented Programming *http://codespeak.net/pypy/dist/pypy/doc/aspect_oriented_programming.html* 2007.

[10] Class decorators *http://www.python.org/dev/peps/pep-3129/* 2008.

[11] Monkey patching *http://en.wikipedia.org/wiki/Monkey_patching* 2008.

[12] Unifying types and classes in Python 2.2 *http://www.python.org/download/releases/2.2.3/descrintro/#metaclasses* 2002.

[13] Whats New in Python 3.0 *http://docs.python.org/dev/3.0/whatsnew/3.0.html#new-class-and-metaclass-stuff* 2008.

[14] Standard Library Reorganization *http://www.python.org/dev/peps/pep-3108/* 2008.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-Oriented Programming.* 1997.

[16] Aspect oriented implementations *http://en.wikipedia.org/wiki/Aspect-oriented_programming#Implementations* 2008.