The Composition Over Inheritance Principle ¶

A principle from the Gang of Four book

Verdict

> In Python as in other programming languages, this grand principle encourages software architects to escape from Object Orientation and enjoy the simpler practices of Object Based programming instead.

This is the second principle propounded by the Gang of Four book at the very beginning, in its "Introduction" chapter. The principle is so important that they display it indented and in italics, like this:

> *Favor object composition over class inheritance.*

Let's take a single design problem and watch how this principle works itself out through several of the classic Gang of Four design patterns. Each design pattern will assemble simple classes, unburdened by inheritance, into an elegant runtime solution.

Problem: the subclass explosion ¶

A crucial weakness of inheritance as a design strategy is that a class often needs to be specialized along several different design axes at once, leading to what the Gang of Four call "a proliferation of classes" in their Bridge chapter and "an explosion of subclasses to support every combination" in their Decorator chapter.

Python's logging module is a good example in the Standard Library itself of a module that follows the Composition Over Inheritance principle, so let's use logging as our example. Imagine a base logging class that has gradually gained subclasses as developers needed to send log messages to new destinations.

```
import sys
import syslog

# The initial class.

class Logger(object):
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()
```

```
# Two more classes, that send messages elsewhere.

class SocketLogger(Logger):
    def __init__(self, sock):
        self.sock = sock

    def log(self, message):
        self.sock.sendall((message + '\n').encode('ascii'))

class SyslogLogger(Logger):
    def __init__(self, priority):
        self.priority = priority

    def log(self, message):
        syslog.syslog(self.priority, message)
```

The problem arises when this first axis of design is joined by another. Let's imagine that log messages now need to be filtered — some users only want to see messages with the word "Error" in them, and a developer responds with a new subclass of `Logger`:

```
# New design direction: filtering messages.

class FilteredLogger(Logger):
    def __init__(self, pattern, file):
        self.pattern = pattern
        super().__init__(file)

    def log(self, message):
        if self.pattern in message:
            super().log(message)

# It works.

f = FilteredLogger('Error', sys.stdout)
f.log('Ignored: this is not important')
f.log('Error: but you want to see this')
```

```
Error: but you want to see this
```

The trap has now been laid, and will be sprung the moment the application needs to filter messages but write them to a socket instead of a file. None of the existing classes covers that case. If the developer plows on ahead with subclassing and creates a `FilteredSocketLogger` that combines the features of both classes, then the subclass explosion is underway.

Maybe the programmer will get lucky and no further combinations will be needed. But in the general case the application will wind up with 3×2=6 classes:

```
Logger            FilteredLogger
SocketLogger      FilteredSocketLogger
SyslogLogger      FilteredSyslogLogger
```

The total number of classes will increase geometrically if *m* and *n* both continue to grow. This is the "proliferation of classes" and "explosion of subclasses" that the Gang of Four want to avoid.

The solution is to recognize that a class responsible for both filtering messages and logging messages is too complicated. In modern Object Oriented practice, it would be accused of violating the “Single Responsibility Principle.”

But how can we distribute the two features of message filtering and message output across different classes?

Solution #1: The Adapter Pattern ¶

One solution is the Adapter Pattern: to decide that the original logger class doesn’t need to be improved, because any mechanism for outputting messages can be wrapped up to look like the file object that the logger is expecting.

1. So we keep the original `Logger`.
2. And we also keep the `FilteredLogger`.
3. But instead of creating destination-specific subclasses, we adapt each destination to the behavior of a file and then pass the adapter to a `Logger` as its output file.

Here are adapters for each of the other two outputs:

```
import socket

class FileLikeSocket:
    def __init__(self, sock):
        self.sock = sock

    def write(self, message_and_newline):
        self.sock.sendall(message_and_newline.encode('ascii'))

    def flush(self):
        pass

class FileLikeSyslog:
    def __init__(self, priority):
        self.priority = priority

    def write(self, message_and_newline):
        message = message_and_newline.rstrip('\n')
        syslog.syslog(self.priority, message)

    def flush(self):
        pass
```

Python encourages duck typing, so an adapter’s only responsibility is to offer the right methods — our adapters, for example, are exempt from the need to inherit from either the classes they wrap or from the `file` type they are imitating. They are also under no obligation to re-implement the full slate of more than a dozen methods that a real file offers. Just as it’s not important that a duck can walk if all you need is a quack, our adapters only need to implement the two file methods that the `Logger` really uses.

And so the subclass explosion is avoided! Logger objects and adapter objects can be freely mixed and matched at runtime without the need to create any further classes:

```
sock1, sock2 = socket.socketpair()

fs = FileLikeSocket(sock1)
logger = FilteredLogger('Error', fs)
logger.log('Warning: message number one')
logger.log('Error: message number two')

print('The socket received: %r' % sock2.recv(512))
```

```
The socket received: b'Error: message number two\n'
```

Note that it was only for the sake of example that the `FileLikeSocket` class is written out above — in real life that adapter comes built-in to Python's Standard Library. Simply call any socket's `makefile()` method to receive a complete adapter that makes the socket look like a file.

Solution #2: The Bridge Pattern ¶

The Bridge Pattern splits a class's behavior between an outer "abstraction" object that the caller sees and an "implementation" object that's wrapped inside. We can apply the Bridge Pattern to our logging example if we make the (perhaps slightly arbitrary) decision that filtering belongs out in the "abstraction" class while output belongs in the "implementation" class.

As in the Adapter case, a separate echelon of classes now governs writing. But instead of having to contort our output classes to match the interface of a Python `file` object — which required the awkward maneuver of adding a newline in the logger that sometimes had to be removed again in the adapter — we now get to define the interface of the wrapped class ourselves.

So let's design the inner "implementation" object to accept a raw message, rather than needing a newline appended, and reduce the interface to only a single method `emit()` instead of also having to support a `flush()` method that was usually a no-op.

```
# The "abstractions" that callers will see.

class Logger(object):
    def __init__(self, handler):
        self.handler = handler

    def log(self, message):
        self.handler.emit(message)

class FilteredLogger(Logger):
    def __init__(self, pattern, handler):
        self.pattern = pattern
        super().__init__(handler)

    def log(self, message):
        if self.pattern in message:
            super().log(message)

# The "implementations" hidden behind the scenes.
```

```
class FileHandler:
    def __init__(self, file):
        self.file = file

    def emit(self, message):
        self.file.write(message + '\n')
        self.file.flush()

class SocketHandler:
    def __init__(self, sock):
        self.sock = sock

    def emit(self, message):
        self.sock.sendall((message + '\n').encode('ascii'))

class SyslogHandler:
    def __init__(self, priority):
        self.priority = priority

    def emit(self, message):
        syslog.syslog(self.priority, message)
```

Abstraction objects and implementation objects can now be freely combined at runtime:

```
handler = FileHandler(sys.stdout)
logger = FilteredLogger('Error', handler)

logger.log('Ignored: this will not be logged')
logger.log('Error: this is important')
```

```
Error: this is important
```

This presents more symmetry than the Adapter. Instead of file output being native to the `Logger` but non-file output requiring an additional class, a functioning logger is now always built by composing an abstraction with an implementation.

Once again, the subclass explosion is avoided because two kinds of class are composed together at runtime without requiring either class to be extended.

Solution #3: The Decorator Pattern ¶

What if we wanted to apply two different filters to the same log? Neither of the above solutions supports multiple filters — say, one filtering by priority and the other matching a keyword.

Look back at the filters defined in the previous section. The reason we cannot stack two filters is that there's an asymmetry between the interface they offer and the interface they wrap: they offer a `log()` method but call their handler's `emit()` method. Wrapping one filter in another would result in an `AttributeError` when the outer filter tried to call the inner filter's `emit()`.

If we instead pivot our filters and handlers to offering the same interface, so that they all alike offer a `log()` method, then we have arrived at the Decorator Pattern:

```
# The loggers all perform real output.

class FileLogger:
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()

class SocketLogger:
    def __init__(self, sock):
        self.sock = sock

    def log(self, message):
        self.sock.sendall((message + '\n').encode('ascii'))

class SyslogLogger:
    def __init__(self, priority):
        self.priority = priority

    def log(self, message):
        syslog.syslog(self.priority, message)

# The filter calls the same method it offers.

class LogFilter:
    def __init__(self, pattern, logger):
        self.pattern = pattern
        self.logger = logger

    def log(self, message):
        if self.pattern in message:
            self.logger.log(message)
```

For the first time, the filtering code has moved outside of any particular logger class. Instead, it's now a stand-alone feature that can be wrapped around any logger we want.

As with our first two solutions, filtering can be combined with output at runtime without building any special combined classes:

```
log1 = FileLogger(sys.stdout)
log2 = LogFilter('Error', log1)

log1.log('Noisy: this logger always produces output')

log2.log('Ignored: this will be filtered out')
log2.log('Error: this is important and gets printed')
```

```
Noisy: this logger always produces output
Error: this is important and gets printed
```

And because Decorator classes are symmetric — they offer exactly the same interface they wrap — we can now stack several different filters atop the same log!

```
log3 = LogFilter('severe', log2)

log3.log('Error: this is bad, but not that bad')
```

```
log3.log('Error: this is pretty severe')
```

```
Error: this is pretty severe
```

But note the one place where the symmetry of this design breaks down: while filters can be stacked, output routines cannot be combined or stacked. Log messages can still only be written to one output.

Solution #4: Beyond the Gang of Four patterns ¶

Python's logging module wanted even more flexibility: not only to support multiple filters, but to support multiple outputs for a single stream of log messages. Based on the design of logging modules in other languages — see PEP 282's "Influences" section for the main inspirations — the Python logging module implements its own Composition Over Inheritance pattern.

1. The `Logger` class that callers interact with doesn't itself implement either filtering or output. Instead, it maintains a list of filters and a list of handlers.
2. For each log message, the logger calls each of its filters. The message is discarded if any filter rejects it.
3. For each log message that's accepted by all the filters, the logger loops over its output handlers and asks every one of them to `emit()` the message.

Or, at least, that's the core of the idea. The Standard Library's logging is in fact more complicated. For example, each handler can carry its own list of filters in addition to those listed by its logger. And each handler also specifies a minimum message "level" like `INFO` or `WARN` that, rather confusingly, is enforced neither by the handler itself nor by any of the handler's filters, but instead by an `if` statement buried deep inside the logger where it loops over the handlers. The total design is thus a bit of a mess.

But we can use the Standard Library logger's basic insight — that a logger's messages might deserve both multiple filters *and* multiple outputs — to decouple filter classes and handler classes entirely:

```
# There is now only one logger.

class Logger:
    def __init__(self, filters, handlers):
        self.filters = filters
        self.handlers = handlers

    def log(self, message):
        if all(f.match(message) for f in self.filters):
            for h in self.handlers:
                h.emit(message)

# Filters now know only about strings!

class TextFilter:
    def __init__(self, pattern):
        self.pattern = pattern
    def match(self, text):
        return self.pattern in text

# Handlers look like "loggers" did in the previous solution.

class FileHandler:
    def __init__(self, file):
        self.file = file

    def emit(self, message):
        self.file.write(message + '\n')
        self.file.flush()

class SocketHandler:
    def __init__(self, sock):
        self.sock = sock

    def emit(self, message):
        self.sock.sendall((message + '\n').encode('ascii'))

class SyslogHandler:
    def __init__(self, priority):
        self.priority = priority

    def emit(self, message):
        syslog.syslog(self.priority, message)
```

Note that only with this final pivot in our design do filters really shine forth with the simplicity they deserve. For the first time, they accept only a string and return only a verdict. All of the previous designs either hid filtering inside one of the logging classes itself, or saddled filters with additional duties beyond simply rendering a verdict.

In fact, the word "log" has dropped entirely away from the name of the filter class, and for a very important reason: there's no longer anything about it that's specific to logging! The `TextFilter` is now entirely reusable in any context that happens to involve strings. Finally decoupled from the specific concept of logging, it will be easier to test and maintain.

Again, as with all Composition Over Inheritance solutions to a problem, classes are composed at runtime without needing any inheritance:

```
f = TextFilter('Error')
h = FileHandler(sys.stdout)
logger = Logger([f], [h])

logger.log('Ignored: this will not be logged')
logger.log('Error: this is important')
```

```
Error: this is important
```

There's a crucial lesson here: design principles like Composition Over Inheritance are, in the end, more important than individual patterns like the Adapter or Decorator. Always follow the principle. But don't always feel constrained to choose a pattern from an official list. The design at which we've now arrived is both more flexible and easier to maintain than any of the previous designs, even though they were based on official Gang of Four patterns but this final design is not. Sometimes, yes, you will find an existing Design Pattern that's a perfect fit for your problem — but if not, your design might be stronger if you move beyond them.

Dodge: "if" statements ¶

I suspect that the above code has startled many readers. To a typical Python programmer, such heavy use of classes might look entirely contrived — an awkward exercise in trying to make old ideas from the 1980s seem relevant to modern Python.

When a new design requirement appears, does the typical Python programmer really go write a new class? No! "Simple is better than complex." Why add a class, when an `if` statement will work instead? A single logger class can gradually accrete conditionals until it handles all the same cases as our previous examples:

```
# Each new feature as an "if" statement.

class Logger:
    def __init__(self, pattern=None, file=None, sock=None,
priority=None):
        self.pattern = pattern
        self.file = file
        self.sock = sock
        self.priority = priority

    def log(self, message):
        if self.pattern is not None:
            if self.pattern not in message:
                return
        if self.file is not None:
            self.file.write(message + '\n')
            self.file.flush()
        if self.sock is not None:
            self.sock.sendall((message + '\n').encode('ascii'))
        if self.priority is not None:
            syslog.syslog(self.priority, message)

# Works just fine.

logger = Logger(pattern='Error', file=sys.stdout)

logger.log('Warning: not that important')
logger.log('Error: this is important')
```

```
Error: this is important
```

You may recognize this example as more typical of the Python design practices you've encountered in real applications.

The `if` statement approach is not entirely without benefit. This class's whole range of possible behaviors can be grasped in a single reading of the code from top to bottom. The parameter list might look verbose but, thanks to Python's optional keyword arguments, most calls to the class won't need to provide all four arguments.

(It’s true that this class can handle only one file and one socket, but that’s an incidental simplification for the sake of readability. We could easily pivot the `file` and `socket` parameters to lists named `files` and `sockets`.)

Given that every Python programmer learns `if` quickly, but can take much longer to understand classes, it might seem a clear win for code to rely on the simplest possible mechanism that will get a feature working. But let’s balance that temptation by making explicit what’s been lost by dodging Composition Over Inheritance:

1. Locality. Reorganizing the code to use `if` statements hasn’t been an unmitigated win for readability. If you are tasked with improving or debugging one particular feature — say, the support for writing to a socket — you will find that you can’t read its code all in one place. The code behind that single feature is scattered between the initializer’s parameter list, the initializer’s code, and the `log()` method itself.
2. Deletability. An underappreciated property of good design is that it makes deleting features easy. Perhaps only veterans of large and mature Python applications will strongly enough appreciate the importance of code deletion to a project’s health. In the case of our class-based solutions, we can trivially delete a feature like logging to a socket by removing the `SocketHandler` class and its unit tests once the application no longer needs it. By contrast, deleting the socket feature from the forest of `if` statements not only requires caution to avoid breaking adjacent code, but raises the awkward question of what to do with the `socket` parameter in the initializer. Can it be removed? Not if we need to keep the list of positional parameters consistent — we would need to retain the parameter, but raise an exception if it’s ever used.
3. Dead code analysis. Related to the previous point is the fact that when we use Composition Over Inheritance, dead code analyzers can trivially detect when the last use of `SocketHandler` in the codebase disappears. But dead code analysis is often helpless to make a determination like “you can now remove all the attributes and `if` statements related to socket output, because no surviving call to the initializer passes anything for `socket` other than `None`.”
4. Testing. One of the strongest signals about code health that our tests provide is how many lines of irrelevant code have to run before reaching the line under test. Testing a feature like logging to a socket is easy if the test can simply spin up a `SocketHandler` instance, pass it a live socket, and ask it to `emit()` a message. No code runs except code relevant to the feature. But testing socket logging in our forest of `if` statements will run at least three times the number of lines of code. Having to set up a logger with the right combination of several features merely to test one of them is an important warning sign, that might seem trivial in this small example but becomes crucial as a system grows larger.
5. Efficiency. I’m deliberately putting this point last, because readability and maintainability are generally more important concerns. But the design problems with the forest of `if` statements are also signalled by the approach’s inefficiency. Even if you want a simple unfiltered log to a single file, every single message will

be forced to run an `if` statement against every possible feature you could have enabled. The technique of composition, by contrast, only runs code for the features you've composed together.

For all of these reasons, I suggest that the apparent simplicity of the `if` statement forest is, from the point of view of software design, largely an illusion. The ability to read the logger top-to-bottom as a single piece of code comes at the cost of several other kinds of conceptual expense that will grow sharply with the size of the codebase.

Dodge: Multiple Inheritance ¶

Some Python projects fall short of practicing Composition Over Inheritance because they are tempted to dodge the principle by means of a controversial feature of the Python language: multiple inheritance.

Let's return to the example code we started with, where `FilteredLogger` and `SocketLogger` were two different subclasses of a base `Logger` class. In a language that only supported single inheritance, a `FilteredSocketLogger` would have to choose to inherit either from `SocketLogger` or `FilteredLogger`, and would then have to duplicate code from the other class.

But Python supports multiple inheritance, so the new `FilteredSocketLogger` can list both `SocketLogger` and `FilteredLogger` as base classes and inherit from both:

```
# Our original example's base class and subclasses.

class Logger(object):
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()

class SocketLogger(Logger):
    def __init__(self, sock):
        self.sock = sock

    def log(self, message):
        self.sock.sendall((message + '\n').encode('ascii'))

class FilteredLogger(Logger):
    def __init__(self, pattern, file):
        self.pattern = pattern
        super().__init__(file)

    def log(self, message):
        if self.pattern in message:
            super().log(message)

# A class derived through multiple inheritance.

class FilteredSocketLogger(FilteredLogger, SocketLogger):
    def __init__(self, pattern, sock):
        FilteredLogger.__init__(self, pattern, None)
        SocketLogger.__init__(self, sock)
```

```
# Works just fine.

logger = FilteredSocketLogger('Error', sock1)
logger.log('Warning: not that important')
logger.log('Error: this is important')

print('The socket received: %r' % sock2.recv(512))
```

```
The socket received: b'Error: this is important\n'
```

This bears several striking resemblances to our Decorator Pattern solution. In both cases:

- There's a logger class for each kind of output (instead of our Adapter's asymmetry between writing files directly but non-files through an adapter).
- The `message` preserves the exact value provided by the caller (instead of our Adapter's habit of replacing it with a file-specific value by appending a newline).
- The filter and loggers are symmetric in that they both implement the same method `log()`. (Our other solutions besides the Decorator had filter classes offering one method and output classes offering another).
- The filter never tries to produce output on its own but, if a message survives filtering, defers the task of output to other code.

These close similarities with our earlier Decorator solution mean that we can compare it with this new code to make an unusually sharp comparison between Composition Over Inheritance and multiple inheritance. Let's sharpen the focus still further with a question:

If we have thorough unit tests for both the logger and filter, how confident are we that they will work together?

1. The success of the Decorator example depends only on the public behaviors of each class: that the `LogFilter` offers a `log()` method that in turn calls `log()` on the object it wraps (which a test can trivially verify using a tiny fake logger), and that each logger offers a working `log()` method. As long as our unit tests verify these two public behaviors, we can't break composition without failing our unit tests.

   Multiple inheritance, by contrast, depends on behavior that cannot be verified by simply instantiating the classes in question. The public behavior of a `FilteredLogger` is that it offers a `log()` method that both filters and writes to a file. But multiple inheritance doesn't merely depend on that public behavior, but on how that behavior is implemented internally. Multiple inheritance will work if the method is deferring to its base class using `super()`, but not if the method does its own `write()` to the file, even though either implementation would satisfy the unit test.

   A test suite must therefore go beyond unit testing and perform actual multiple inheritance on the class — or else monkey patch to verify that `log()` calls `super().log()` — to guarantee that multiple inheritance keeps working as future developers work on the code.

2. Multiple inheritance has introduced a new `__init__()` method because neither base class's `__init__()` method accepts enough arguments for a combined filter and logger. That new code needs to be tested, so at least one test will be necessary for every new subclass.

   You might be tempted to concoct a scheme to avoid a new `__init__()` for every subclass, like accepting `*args` and then passing them on to `super().__init__()`. (If you do pursue that approach, review the classic essay "Python's Super Considered Harmful" which argues that only `**kw` is in fact safe.) The problem with such a scheme is that it hurts readability — you can no longer figure out what arguments an `__init__()` method takes simply by reading its parameter list. And type checking tools will no longer be able to guarantee correctness.

   But whether you give each derived class its own `__init__()` or design them to chain together, your unit tests of the original `FilteredLogger` and `SocketLogger` can't by themselves guarantee that the classes initialize correctly when combined.

   By contrast, the Decorator's design leaves its initializers happily and strictly orthogonal. The filter accepts its `pattern`, the logger accepts its `sock`, and there is no possible conflict between the two.

3. Finally, it's possible that two classes work fine on their own, but have class or instance attributes with the same name that will collide when the classes are combined through multiple inheritance.

   Yes, our small examples here make the chance of collision look too small to worry about — but remember that these examples are merely standing in for the vastly more complicated classes you might write in real applications.

   Whether the programmer writes tests to guard against collision by running `dir()` on instances of each class and checking for attributes they have in common, or by writing an integration test for every possible subclass, the original unit tests of the two separate classes will once again have failed to guarantee that they can combine cleanly through multiple inheritance.

For any of these reasons, the unit tests of two base classes can stay green even as their ability to be combined through multiple inheritance is broken. This means that the Gang of Four's "explosion of subclasses to support every combination" will also afflict your tests. Only by testing every combination of *m*×*n* base classes in your application can you make it safe for the application to use such classes at runtime.

In addition to breaking the guarantees of unit testing, multiple inheritance involves at least three further liabilities.

4. Introspection is simple in the Decorator case. Simply `print(my_filter.logger)` or view that attribute in a debugger to see what sort of output logger is attached. In the case of multiple inheritance, however, you can only learn which filter and logger have been combined by examining the metadata of the class itself — either by reading its `__mro__` or subjecting the object to a series of `isinstance()` tests.
5. It's trivial in the Decorator case to take a live combination of a filter and logger and at runtime to swap in a different logger through assignment to the `.logger` attribute — say, because the user has just toggled a preference in the application's interface. But to do the same in the multiple inheritance case would require the rather more objectionable maneuver of overwriting the object's class. While changing an object's class at runtime is not impossible in a dynamic language like Python, it's generally considered a symptom that software design has gone wrong.
6. Finally, multiple inheritance provides no built-in mechanism to help the programmer order the base classes correctly. The `FilteredSocketLogger` won't successfully write to a socket if its base classes are swapped and, as dozens of Stack Overflow questions attest, Python programmers have perpetual difficultly with putting third-party base classes in the right order. The Decorator pattern, by contrast, makes it obvious which way the classes compose: the filter's `__init__()` wants a `logger` object, but the logger's `__init__()` doesn't ask for a `filter`.

Multiple inheritance, then, incurs a number of liabilities without adding a single advantage. At least in this example, solving a design problem with inheritance is strictly worse than a design based on composition.

Dodge: Mixins ¶

The `FilteredSocketLogger` in the previous section needed its own custom `__init__()` method because it needed to accept arguments for both of its base classes. But it turns out that this liability can be avoided. Of course, in cases where a subclass doesn't require any extra data, the problem doesn't arise. But even classes that do require extra data can have it delivered by other means.

We can make the `FilteredLogger` more friendly to multiple inheritance if we provide a default value for `pattern` in the class itself and then invite callers to customize the attribute directly, out-of-band of initialization:

```
# Don't accept a "pattern" during initialization.

class FilteredLogger(Logger):
    pattern = ''

    def log(self, message):
        if self.pattern in message:
            super().log(message)
```

```
# Multiple inheritance is now simpler.

class FilteredSocketLogger(FilteredLogger, SocketLogger):
    pass  # This subclass needs no extra code!

# The caller can just set "pattern" directly.

logger = FilteredSocketLogger(sock1)
logger.pattern = 'Error'

# Works just fine.

logger.log('Warning: not that important')
logger.log('Error: this is important')

print('The socket received: %r' % sock2.recv(512))
```

```
The socket received: b'Error: this is important\n'
```

Having pivoted the `FilteredLogger` to an initialization maneuver that's orthogonal to that of its base class, why not push the idea of orthogonality to its logical conclusion? We can convert the `FilteredLogger` to a "mixin" that lives entirely outside the class hierarchy with which multiple inheritance will combine it.

```
# Simplify the filter by making it a mixin.

class FilterMixin:  # No base class!
    pattern = ''

    def log(self, message):
        if self.pattern in message:
            super().log(message)

# Multiple inheritance looks the same as above.

class FilteredLogger(FilterMixin, FileLogger):
    pass  # Again, the subclass needs no extra code.

# Works just fine.

logger = FilteredLogger(sys.stdout)
logger.pattern = 'Error'
logger.log('Warning: not that important')
logger.log('Error: this is important')
```

```
Error: this is important
```

The mixin is conceptually simpler than the filtered subclass we saw in the last section: it has no base class that might complicate method resolution order, so `super()` will always call the next base class listed in the `class` statement.

A mixin also has a simpler testing story than the equivalent subclass. Whereas the `FilteredLogger` would need tests that both run it standalone and also combine it with other classes, the `FilterMixin` only needs tests that combine it with a logger. Because the mixin is by itself incomplete, a test can't even be written that runs it standalone.

But all the other liabilities of multiple inheritance still apply. So while the mixin pattern does improve the readability and conceptual simplicity of multiple inheritance, it's not a complete solution for its problems.

Dodge: Building classes dynamically ¶

As we saw in the previous two sections, neither traditional multiple inheritance nor mixins solve the Gang of Four's problem of "an explosion of subclasses to support every combination" — they merely avoid code duplication when two classes need to be combined.

Multiple inheritance still requires, in the general case, "a proliferation of classes" with *m*×*n* class statements that each look like:

```
class FilteredSocketLogger(FilteredLogger, SocketLogger):
    ...
```

But it turns out that Python offers a workaround.

Imagine that our application reads a configuration file to learn the log filter and log destination it should use, a file whose contents aren't known until runtime. Instead of building all *m*×*n* possible classes ahead of time and then selecting the right one, we can wait and take advantage of the fact that Python not only supports the `class` statement but a builtin `type()` function that creates new classes dynamically at runtime:

```
# Imagine 2 filtered loggers and 3 output loggers.

filters = {
    'pattern': PatternFilteredLog,
    'severity': SeverityFilteredLog,
}
outputs = {
    'file': FileLog,
    'socket': SocketLog,
    'syslog': SyslogLog,
}

# Select the two classes we want to combine.

with open('config') as f:
    filter_name, output_name = f.read().split()

filter_cls = filters[filter_name]
output_cls = outputs[output_name]

# Build a new derived class (!)

name = filter_name.title() + output_name.title() + 'Log'
cls = type(name, (filter_cls, output_cls), {})

# Call it as usual to produce an instance.

logger = cls(...)
```

The tuple of classes passed to `type()` has the same meaning as the series of base classes in a `class` statement. The `type()` call above creates a new class through multiple inheritance from both a filtered logger and an output logger.

Before you ask: yes, it would also work to build a `class` statement as plain text and then pass it to `eval()`.

But building classes on-the-fly carries severe liabilities.

- Readability suffers. A human reading the above snippet of code will have to do extra work to determine what sort of object an instance of `cls` is. Also, many Python programmers aren't familiar with `type()` and will need to stop and puzzle over its documentation. If they have difficulty with the novel concept that classes can be defined dynamically, they might still be confused.
- If a constructed class like `PatternFilteredFileLog` is named in an exception or error message, the developer will probably be unhappy to discover that nothing comes up when they search the code for that class name. Debugging becomes more difficult when you cannot even locate a class. Considerable time may be spent searching the codebase for `type()` calls and trying to determine which one generated the class. Sometimes developers have to resort to calling each method with bad arguments and using the line numbers in the resulting tracebacks to track down the base classes.
- Type introspection will, in the general case, fail for classes constructed dynamically at runtime. "Jump to class" shortcuts in your editor won't have anywhere to take you when you highlight an instance of `PatternFilteredFileLog` in the debugger. And type checking engines like mypy and pyre-check will be unlikely to offer the strong protections for your generated class that they're able to provide for normal Python classes.
- The beautiful Jupyter Notebook feature `%autoreload` possesses a nearly preternatural ability to detect and reload modified source code in a live Python interpreter. But it's foiled, for example, by the multiple inheritance classes that matplotlib builds at runtime through `type()` calls inside its `subplot_class_factory()`.

Once its liabilities are weighed, the attempt to use runtime class generation as a last-ditch maneuver to rescue the already faulty mechanism of multiple inheritance stands as a *reductio ad absurdum* of the entire project of dodging Composition Over Inheritance when you need an object's behavior to vary over several independent axes.