As Alan Kay wrote, inheritance is used for different reasons, and multiple inheritance adds alternatives and complexity. It's easy to create incomprehensible and brittle designs using multiple inheritance. Since we don't have a comprehensive theory, here are a few tips to avoid spaghetti class graphs.

1. Distinguish interface inheritance from implementation inheritance

When dealing with multiple inheritance it's useful to keep straight the reasons why subclassing is done in the first place. The main reasons are:

- Inheritance of interface: creates a sub-type, implying an "is-a" relationship.
- Inheritance of implementation: avoids code duplication by reuse.

In practice both uses are often simultaneous, but whenever you can make the intent clear, do it. Inheritance for code reuse is an implementation detail, and it can often be replaced by composition and delegation. On the other hand, interface inheritance is the backbone of a framework.

2. Make interfaces explicit with ABCs

In modern Python, if a class is designed to define an interface, it should be an explicit ABC. In Python ≥ 3.4 this means: subclass `abc.ABC` or another ABC (see "ABC syntax details" on page 330 if you need to support older Python versions).

3. Use mixins for code reuse

If a class is designed to provide method implementations for reuse by multiple unrelated subclasses, without implying an "is-a" relationship, it should be an explicit *mixin class*. Conceptually, a mixin does not define a new type, it merely bundles methods for reuse. A mixin should never be instantiated, and concrete classes should not inherit only from a mixin. Each mixin should provide a single specific behavior, implementing few and very closely related methods.

4. Make mixins explicit by naming

There is no formal way in Python to state that a class is a mixin, so it is highly recommended that they are named with a `…Mixin` suffix. Tkinter does not follow this advice, but if it did, `XView`, would be `XViewMixin`, `Pack` would be `PackMixin` and so on with all the classes where I put the «mixin» tag Figure 12-3.

5. An ABC may also be a mixin; the reverse is not true

Since an ABC can implement concrete methods, it works as a mixin as well. An ABC also defines a type, which a mixin does not. And an ABC can be the sole base class of

any another class, while a mixin should never be subclassed alone except by another, more specialized mixin — not a common arrangement in real code.

One restriction applies to ABCs and not to mixins: the concrete methods implemented in an ABC should only collaborate with methods of the same ABC and its superclasses. This implies that concrete methods in an ABC are always for convenience, because everything they do an user of the class can also do by calling other methods of the ABC.

6. Don't subclass from more than one concrete class

Concrete classes should have zero or at most one concrete superclass[6]. In other words, all but one of the superclasses of a concrete class should be ABCs or mixins. For example, in the code below, if `Alpha` is a concrete class, then `Beta` and `Gamma` must be ABCs or mixins:

```
class MyConcreteClass(Alpha, Beta, Gamma):
    """This is a concrete class: it can be instantiated."""
    # ... more code ...
```

7. Provide aggregate classes to users

If some combination of ABCs or mixins is particularly useful to client code, provide a class that brings them together in a sensible way. Grady Booch calls this an *aggregate class*.[7]

For example, here is the complete source code for `tkinter.Widget`:

```
class Widget(BaseWidget, Pack, Place, Grid):
    """Internal class.

    Base class for a widget which can be positioned with the
    geometry managers Pack, Place or Grid."""
    pass
```

The body of `Widget` is empty, but the class provides a useful service: it brings together four superclasses so that anyone who needs to create a new widget does not need remember all those mixins, or wonder if they need to be declared in a certain order in a `class` statement. A better example of this is the Django `ListView` class, which we'll discuss shortly, in "A modern example: mixins in Django generic views" on page 364.

6. In "Waterfowl and ABCs" on page 316, Alex Martelli quotes Scott Meyer's *More Effective C++* which goes even further: "all non-leaf classes should be abstract" i.e. concrete classes should not have concrete superclasses at all.

7. "A class that is constructed primarily by inheriting from mixins and does not add its own structure or behavior is called an *aggregate class*.", Grady Booch et.al. — *Object Oriented Analysis and Design, 3e* (Addison-Wesley, 2007), p. 109.

8. "Favor object composition over class inheritance."

This quote comes straight the *Design Patterns* book[8], and is the best advice I can offer here. Once you get comfortable with inheritance, it's too easy to overuse it. Placing objects in a neat hierarchy appeals to our sense of order; programmers do it just for fun.

However, favoring composition leads to more flexible designs. For example, in the case of the `tkinter.Widget` class, instead of inheriting the methods from all geometry managers, widget instances could hold a reference to a geometry manager, and invoke its methods. After all, a `Widget` should not "be" a geometry manager, but could use the services of one via delegation. Then you could add a new geometry manager without touching the widget class hierarchy and without worrying about name clashes. Even with single inheritance, this principle enhances flexibility, because subclassing is a form of tight coupling, and tall inheritance trees tend to be brittle.

Composition and delegation can replace the use of mixins to make behaviors available to different classes, but cannot replace the use of interface inheritance to define a hierarchy of types.

We will now analyze Tkinter from the point of view of these recommendations.

Tkinter: the good, the bad and the ugly

Keep in mind that Tkinter has been part of the Standard Library since Python 1.1 was released in 1994. Tkinter is a layer on top of the excellent Tk GUI toolkit of the Tcl language. The Tcl/Tk combo is not originally object oriented, so the Tk API is basically a vast catalog of functions. However, the toolkit is very object oriented in its concepts, if not in its implementation.

Most advice in the previous section is not followed by Tkinter, with #7 being a notable exception. Even then, it's not a great example, because composition would probably work better for integrating the geometry managers into `Widget`, as discussed in #8.

The docstring of `tkinter.Widget` starts with the words "Internal class." This suggests that `Widget` should probably be an ABC. Although it has no methods of its own, `Widget` does define an interface. Its message is: "You can count on every Tkinter widget providing basic widget methods (`__init__`, `destroy`, and dozens of Tk API functions), in addition to the methods of all three geometry managers." We can agree that this is not a great interface definition (it's just too broad), but it is an interface, and `Widget` "defines" it as the union of the interfaces of its superclasses.

8. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, *Introduction*, p. 20.