PYTHON PROGRAMMING *with* DESIGN PATTERNS

JAMES W. COOPER

Python Programming with Design Patterns

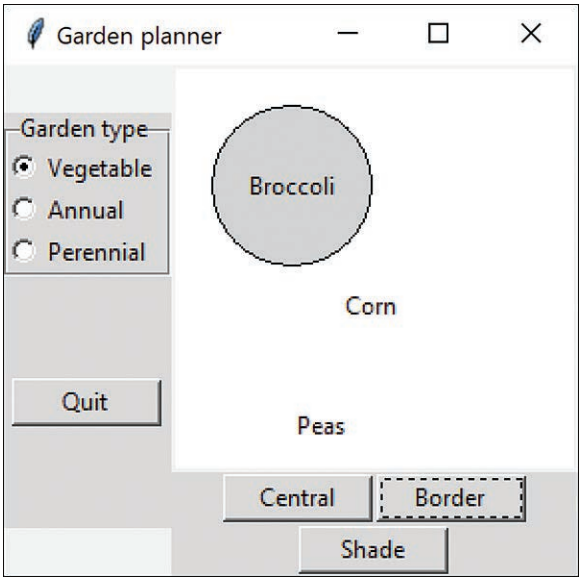

Figure 7-1 Garden planner interface

How the User Interface Works

This simple interface consists of two parts: On the left side, you select the garden type; on the right side, you select the plant category. When you click one of the garden types, this actuates the Abstract Factory and copies the correct garden into the `Gardener` class. Then when a user clicks one of the plant type buttons, the plant type is returned and the name of that plant is displayed.

One of the great strengths of the Abstract Factory is that you can easily add new subclasses. For example, if you needed a `GrassGarden` or a `WildFlowerGarden`, you can subclass `Garden` and produce these classes. The only real change you'd need to make in any existing code is to add some way to choose these new kinds of gardens.

Consequences of the Abstract Factory Pattern

One of the main purposes of the Abstract Factory is that it isolates the concrete classes that are generated. The actual class names of these classes are hidden in the factory and need not be known at the client level.

Because of this isolation of classes, you can freely change or interchange these product class families. Further, because you generate only one kind of concrete class, this system keeps you from inadvertently using classes from different families of products. However, adding new class families takes some effort because you need to define new, unambiguous conditions that cause such a new family of classes to be returned.

Although all the classes that the Abstract Factory generates have the same base class, there is nothing to prevent some subclasses from having additional methods that differ from the methods of other classes. For example, a `BonsaiGarden` class might have a `Height` or `WateringFrequency` method that does not exist in other classes. This presents the same problem as occurs in any subclasses: you don't know whether you can call a class method unless you know whether that subclass is one that allows those methods. This problem has the same two solutions as in any similar case: Either you can define all the methods in the base class, even if they don't always have an actual function, or you can test to see which kind of class you have.

Thought Questions

If you are writing a program to track investments, such as stocks, bonds, metal futures, and derivatives, how might you use an Abstract Factory?

Code on GitHub

The program Gardening.py launches the user interface shown in this chapter, along with exercises for the Abstract Factory and the various `Garden` classes.

8

The Singleton Pattern

The Singleton pattern is grouped with the other creational patterns, although, to some extent, it is a "noncreational" pattern. There are any number of cases in programming where you need to make sure that there can be one, and only one, instance of a class. For example, your system can have only one window manager or print spooler, as well as a single point of access to a database engine.

Python does not directly have a feature where a single static variable will be accessible from all class instances, so a simple flag won't work. Instead, this technique utilizes two subtle Python features you might not be aware of: the `static` method and the `__instance` variable.

Python has a Decorator that tells the compiler to make only a single static method within a class:

```
@staticmethod
```

This makes the method that follows static, rather than one that has a fresh copy for each instance of the class. The top of this class is written like this:

```
class Singleton:
    __instance = None

    # static method declared here
    @staticmethod
    def getInstance():
        if Singleton.__instance == None:
            Singleton()
        return Singleton.__instance
```

Basically, it says that if the `__instance` variable is `None`, create an instance of `Singleton`.

Because constructors do not return values, the problem is how to find out whether creating an instance was successful.

The best way, proposed on the tutorialspoint.com website, is to create a class that throws an `Exception` when it is instantiated more than once. Let's create our own `Exception` class for this case:

```
class SingletonException(Exception):
    def __init__(self, message):
        # Call the base class constructor
             # with the parameters it needs
        super().__init__(message)
```

Note that, other than calling its parent classes through the `super()` method, this new exception type doesn't do anything in particular. However, it is still convenient to have our own named exception type so that the compiler will warn us of the type of exception we must catch when we attempt to `create` an instance of `PrintSpooler` or anything else we create as a Singleton.

Throwing the Exception

The rest of the Singleton is just the `__init__` method, where the class is first created.

```
def __init__(self, name):
    if Singleton.__instance != None:
        raise SingletonException(
               "This class is a singleton!")
    else:
        Singleton.__instance = self
        self.name = name
        print("creating: "+ name)
```

If there is no instance of Singleton yet, it creates one and stores it in the `__instance` variable. If there already is an instance, we raise the `SingletonException`.

Creating an Instance of the Class

Now that we've created a simple Singleton pattern in the eponymous class, let's see how to use it. Remember that we must enclose every method that may throw an exception in a `try – except` block.

```
try:
    al = Singleton("Alan")
    bo = Singleton("Bob")
except SingletonException as e:
    print("two instances of a Singleton")
    details = e.args[0]
    print(details)
else:
    print (al.getName())
    print(bo.getName())
```

Then, if we execute this program, we get the following two messages:

```
creating: Alan
two instances of a Singleton
This class is a singleton!
```

where the last two lines indicate that an exception was thrown as expected. One message was generated to catch the exception, and the other message was provided by the Singleton itself.

One advantage of this approach is that you can restrict the singleton to a small number of instances bigger than 1 without extensive reprogramming (if you can think of a reason for that approach).

Static Classes As Singleton Patterns

There already is a kind of Singleton class in the standard Python class libraries: the `math` class. This class has all methods declared as `@staticmethod`, meaning that the class cannot be extended. The purpose of the `math` class is to wrap common mathematical functions, such as `sin` and `log`, in a class-like structure since the Python language does not support functions that are not methods in a class.

You can use the same approach to a Singleton Spooler pattern, giving it a static method. You can't create *any* instance of classes such as `math` or this Spooler, and can only call the static methods directly in the existing final class.

```
class Spooler:
    @staticmethod
    def printit(text):
        print(text) # simulate printing
name = "Fred"
Spooler.printit(name)
```

Note that we now invoke the Spooler `printit` method directly as `Spooler.printit`.

Finding the Singletons in a Large Program

In a large, complex program, it may not be simple to discover where in the code a Singleton has been instantiated.

One solution is to create such singletons at the beginning of the program and pass them as arguments to the major classes that might need to use them.

```
pr1 = iSpooler.Instance()
cust = Customers(pr1)
```

A more elaborate solution could be to create a registry of all the Singleton classes in the program and make the registry generally available. Each time a Singleton instantiates itself, it notes that in the registry. Then any part of the program can ask for the instance of any singleton using an identifying string and get back that instance variable.

The disadvantage of the registry approach is that type checking may be reduced, since the table of singletons in the registry probably keeps all the singletons as Objects (for example, in a `Hashtable` object). Of course, the registry itself is probably a singleton and must be passed to all parts of the program using the constructor or various set functions.

Other Consequences of the Singleton Pattern

Consequences of the Singleton pattern include the following:

1. It can be difficult to subclass a Singleton because this can work only if the base Singleton class has not yet been instantiated.
2. You can easily change a Singleton to allow a small number of instances in which this is allowable and meaningful.

Sample Code on GitHub

- Spooler.py: Prototype of Spooler
- Testlock.py: Creates a single instance of a Singleton