CHAPTER 16

The Singleton Pattern and the Double-Checked Locking Pattern

Overview

This chapter continues the e-tailing case study discussed in Chapter 14, "The Strategy Pattern" and Chapter 15, "The Decorator Pattern."

In this chapter

In this chapter,

- I introduce the Singleton pattern.
- I describe the key features of the Singleton pattern.
- I introduce a variant to the Singleton called the Double-Checked Locking pattern.
- I describe some of my experiences using the Singleton pattern in practice.

The Singleton pattern and the Double-Checked Locking pattern are very simple and very common. Both are used to ensure that only one object of a particular class is instantiated. The distinction between the patterns is that the Singleton pattern is used in single-threaded applications while the Double-Checked Locking pattern is used in multithreaded applications.[1]

1. If you do not know what a multithreaded application is, don't worry; you need only concern yourself with the Singleton pattern at this time.

Introducing the Singleton Pattern

The intent, according to the Gang of Four

According to the Gang of Four, the Singleton's intent is to

> Ensure a class only has one instance, and provide a global point of access to it.[2]

How the Singleton pattern works

The Singleton pattern works by having a special method that is used to instantiate the desired object.

> When this method is called, it checks to see if the object has already been instantiated. If it has, the method simply returns a reference to the object. If not, the method instantiates it and returns a reference to the new instance.
>
> To ensure that this is the only way to instantiate an object of this type, I define the constructor of this class to be protected or private.

Applying the Singleton Pattern to the Case Study

A motivating example: instantiate tax calculation strategies only once and only when needed

In Chapter 14, I encapsulated the rules about taxes within strategy objects. I have to derive a CalcTax class for each possible tax calculation rule. This means that I need to use the same objects over and over again, just alternating between their uses.

For performance reasons, I might not want to keep instantiating them and throwing them away again and again. And, while I could instantiate all of the possible strategies at the start, this could

2. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, Mass.: Addison-Wesley, 1995, p. 127.

become inefficient if the number of strategies grew large. (Remember, I may have many other strategies throughout my application.) Instead, it would be best to instantiate them as needed, but only do the instantiation once.

The problem is that I do not want to create a separate object to keep track of what I have already instantiated. Rather, I would like the objects themselves (that is, the strategies) be responsible for handling their own single instantiation.

This is the purpose of the Singleton pattern. It allows me to instantiate an object only once, without requiring the client objects to be concerned with whether it already exists or not.

Singleton makes objects responsible for themselves

The Singleton could be implemented in code as shown in Example 16-1. In this example, I create a method *(getInstance)* that will instantiate at most one USTax object. The Singleton protects against someone else instantiating the USTax object directly by making the constructor private, which means that no other object can access it.

Example 16-1 Java Code Fragment: Singleton Pattern

```
class USTax  {
  private static USTax instance;
  private USTax():
  public static USTax getlnstance () { if
    (instance== null)
   instance= new USTax(); return
instance; } }
```

The Singleton Pattern: Key Features

Intent	You want to have only *one* of an object but there is no global object that controls the instantiation of this object.
Problem	Several different client objects need to refer to the same thing and you want to ensure that you do not have more than one of them.
Solution	Guarantees one instance.
Participants and Collaborators	Clients create an get Instance of the Singleton solely through the instance method.
Consequences	Clients need not concern themselves whether an instance of the Singleton exists. This can be controlled from within the Singleton.
Implementation	• Add a private static member of the class that refers to the desired object (initially, it is NULL). • Add a public static method that instantiates this class if this member is NULL (and sets this member's value) and then returns the value of this member. • Set the constructor's status to protected or private so that no one can directly instantiate this class and bypass the static constructor mechanism.
GoF Reference	Pages 127-134.

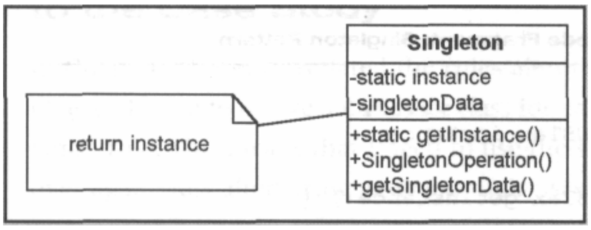

Figure 16-1 Standard, simplified view of the Singleton pattern.

A Variant: The Double-Checked Locking Pattern

Only for multithreaded applications

This pattern *only* applies to multithreaded applications. If you are not involved with multithreaded applications you might want to skip this section. This section assumes that you have a basic understanding of multithreaded issues, including synchronization.

A problem with the Singleton pattern may arise in multithreaded applications.

In a multithreaded mode, Singleton does not always work properly

Suppose two calls to *getInstance ()* are made at exactly the same time. This can be very bad. Consider what can happen in this case:

1. The first thread checks to see if the instance exists. It does not, so it goes into the part of the code that will create the first instance.
2. However, before it has done that, suppose a second thread also looks to see if the instance member is NULL. Since the first thread hasn't created anything yet, the instance is still equal to NULL, so the second thread also goes into the code that will create an object.
3. Both threads now perform a *new* on the Singleton object, thereby creating two objects.

Is this a problem? It may or may not be.

None, small, bad, or worse

- If the Singleton is absolutely stateless, this may not be a problem.
- In Java, the problem will simply be that we are taking up an extra bit of memory.
- In C++, the program may create a memory leak, since it will only delete one of the objects when I have created two of them.
- If the Singleton object has some state, subtle errors can creep in. For example,
  - If the object creates a connection, there will actually be two connections (one for each object).
  - If a counter is used, there will be two counters.

It may be very difficult to find these problems. First of all, the dual creation is very intermittent—it usually won't happen. Second, it may not be obvious why the counts are off, as only one client object will contain one of the Singleton objects while all of the other client objects will refer to the other Singleton.

Synchronizing the creation of the Singleton object

At first, it appears that all I need to do is synchronize the test for whether the Singleton object has been created. The only problem is that this synchronization may end up being a severe bottleneck, because all of the threads will have to wait for the check on whether the object already exists.

Perhaps instead, I could put some synchronization code in after the if (instance== null) test. This will not work either. Since it would be possible that both calls could meet the NULL test and then attempt to synchronize, I could still end up making two Singleton objects, making them one at a time.

A simple solution: double-checked locking

The solution is to do a "synch" after the test for NULL and then check again to make sure the instance member has not yet been created. I show this in Example 16-2. This is called *double-checked locking.*[3] The intent is to optimize away unnecessary locking. This synchronization check happens at most one time, so it will not be a bottleneck.

The features of double-checked locking are as follows:

- Unnecessary locking is avoided by wrapping the call to *new* with another conditional test.
- Support for multithreaded environments.

3. Martin, R., Riehle, D., Buschmann, R, *Pattern Language of Program Design,* Reading, Mass.: Addison-Wesley, 1998, p. 363.

Example 16-2 Java Code Fragment: Instantiation Only

```
class USTax extends  CalcTax
  { private USTax  instance; private
  USTax  ()  {
  }
  private synchronized static
    void doSync () {
    // just here for sync
  }
  public USTax getInstance() { if
    (instance== null)
    { USTax.doSync(); if
    (instance== null) instance=
    new USTax();
    } return
  instance;
  }
 }
```

Field Notes: Using the Singleton and Double-Checked Locking Patterns

Only use when needed

If you know you are going to need an object and no performance issue requires you to defer instantiation of the object until it's needed, it is usually simpler to have a static member contain a reference to the object.

Typically stateless

In multithreaded applications, Singletons typically have to be thread safe (because the single object may be shared by multiple objects). This means having no data members but using only variables whose scope is no larger than a method.

Summary

In this chapter

The Singleton and Double-Checked Locking patterns are common patterns to use when you want to ensure that there is only one instance of an object. The Singleton is used in single-threaded applications while the Double-Checked Locking pattern is used in multithreaded applications.

Supplement: C++ Code Examples

Example 16-3 C++ Code Fragment: Singleton Pattern

```
Class USTax
  { public:
    static USTax* getlnstance();
  private:
    USTax(};
    static USTax* instance;
}
USTax::USTax ()
  { instance= 0;
}
USTax* USTax::getlnstance () { if
  (instance== 0) {
     instance= new USTax;
  } return
instance;
}
```

Example 16-4 C++ Code Fragment: Double-Checked Locking Pattern

```
class USTax : public CalcTax
  { public:
    static USTax* getlnstance(};
  private:
    USTax();
    static USTax* instance;
};
USTax::USTax ()
  { instance= 0;
}
USTax* USTax::getlnstance () { if
  (instance== 0) {
    // do sync here
    if (instance== 0) {
    }
  }
  return instance;
}
```